Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTE UNIVERSITE GRENOBLE ALPES

Spécialité : **Nanoélectronique et nanotechnologies**

Arrêté ministériel : 7 août 2006

Présentée par

Thiago RAUPP DA ROSA

Thèse dirigée par **Fabien CLERMIDY** et
codirigée par **Romain LEMAIRE**

préparée au sein du **Laboratoire Intégration Silicium des Architectures Numériques**
dans **l'École Doctorale Électronique, Électrotechnique, Automatique & Traitement du Signal (EEATS)**

Support des communications dans des architectures multicœurs par l'intermédiaire de mécanismes matériels et d'interfaces de programmation standardisées

Thèse soutenue publiquement le **08 avril 2016**,
devant le jury composé de :

M. Frédéric ROUSSEAU
Professeur, Université Grenoble Alpes, TIMA, Président
M. Gilles SASSATELLI
Directeur de Recherche, HDR, LIRMM, Rapporteur
M. Sébastien PILLEMENT
Professeur, École Polytechnique de l'Université de Nantes, Rapporteur
M. Fabien CLERMIDY
Docteur, HDR, CEA-LETI, Directeur de thèse
M. Romain LEMAIRE
Docteur, CEA-LETI, Co-encadrant de thèse, Invité

UNIVERSITE GRENOBLE ALPES

Abstract

Communication Support in Multi-core Architectures through Hardware Mechanisms and Standardized Programming Interfaces

The application constraints driving the design of embedded systems are constantly demanding higher performance and power efficiency. To meet these constraints, current SoC platforms rely on replicating several processing cores while adding dedicated hardware accelerators to handle specific tasks. However, developing embedded applications is becoming a key challenge, since applications workload will continue to grow and the software technologies are not evolving as fast as hardware architectures, leaving a gap in the full system design. Indeed, the increased programming complexity can be associated to the lack of software standards that supports heterogeneity, frequently leading to custom solutions. On the other hand, implementing a standard software solution for embedded systems might induce significant performance and memory usage overheads. Therefore, this Thesis focus on decreasing this gap by implementing hardware mechanisms in co-design with a standard programming interface for embedded systems. The main objectives are to increase programmability through the implementation of a standardized communication application programming interface (MCAPI), and decrease the overheads imposed by the software implementation through the use of the developed hardware mechanisms.

The contributions of the Thesis comprise the implementation of MCAPI for a generic multi-core platform and dedicated hardware mechanisms to improve communication connection phase and overall performance of data transfer phase. It is demonstrated that the proposed mechanisms can be exploited by the software implementation without increasing software complexity. Furthermore, performance estimations obtained using a SystemC/TLM simulation model for the reference multi-core architecture show that the proposed mechanisms provide significant gains in terms of latency (up to 97%), throughput (40x increase) and network traffic (up to 68%) while reducing processor workload for both characterization test-cases and real application benchmarks.

UNIVERSITE GRENOBLE ALPES

Résumé

Support des communications dans des architectures multicœurs par l'intermédiaire de mécanismes matériels et d'interfaces de programmation standardisées

L'évolution des contraintes applicatives imposent des améliorations continues sur les performances et l'efficacité énergétique des systèmes embarqués. Pour répondre à ces contraintes, les plateformes « SoC » actuelles s'appuient sur la multiplication des cœurs de calcul, tout en ajoutant des accélérateurs matériels dédiés pour gérer des tâches spécifiques. Dans ce contexte, développer des applications embarquées devient un défi complexe, en effet la charge de travail des applications continue à croître alors que les technologies logicielles n'évoluent pas aussi vite que les architectures matérielles, laissant un écart dans la conception complète du système. De fait, la complexité accrue de programmation peut être associée à l'absence de standards logiciels qui prennent en charge l'hétérogénéité des architectures, menant souvent à des solutions ad hoc. A l'opposé, l'utilisation d'une solution logicielle standardisée pour les systèmes embarqués peut induire des surcoûts importants concernant les performances et l'occupation de la mémoire si elle n'est pas adaptée à l'architecture. Par conséquent, le travail de cette Thèse se concentre sur la réduction de cet écart en mettant en œuvre des mécanismes matériels dont la conception prend en compte une interface de programmation standard pour systèmes embarqués. Les principaux objectifs sont ainsi d'accroître la programmabilité par la mise en œuvre d'une interface de programmation : MCAPI, et de diminuer la charge logiciel des cœurs grâce à l'utilisation des mécanismes matériels développés.

Les contributions de la thèse comprennent la mise en œuvre de MCAPI pour une plate-forme multicœur générique et des mécanismes matériels pour améliorer la performance globale de la configuration de la communication et des transferts de données. Il est démontré que les mécanismes peuvent être pris en charge par les interfaces logicielles sans augmenter leur complexité. En outre, les résultats de performance obtenus en utilisant un modèle SystemC/TLM de l'architecture multicœurs de référence montrent que les mécanismes proposés apportent des gains significatifs en termes de latence, débit, trafic réseau, temps de charge processeur et temps de communication sur des cas d'étude et des applications complètes.

Contents

List of Figures

List of Tables

Abbreviations

API	**A**pplication **P**rogramming **I**nterface
APU	**A**ccelerated **P**rocessing **U**nit
BMM	**B**uffer **M**anager **M**echanism
COBRA	**Co**gnitive **B**aseband **Ra**dio
CPU	**C**entral **P**rocessing **U**nit
CSDF	**C**yclo-**S**tatic **D**ata **F**low
DMA	**D**irect **M**emory **A**ccess
DSP	**D**igital **S**ignal **P**rocessing
DVFS	**D**ynamic **V**oltage and **F**requency **S**caling
EDGE	**E**nhanced **D**ata Rate for **G**SM **E**volution
ES	**E**vent **S**ynchronizer
FIFO	**F**irst-**I**n **F**irst-**O**ut
FFT	**F**ast **F**ourier **T**ransform
FPGA	**F**ield-**P**rogrammable **G**ate **A**rray
FSM	**F**inite **S**tate **M**achine
GALS	**G**lobaly **A**synchronous **L**ocally **S**ynchronous
GPP	**G**eneral **P**urpose **P**rocessor
GPU	**G**raphics **P**rocessing **U**nit
GSM	**G**lobal **S**ystem for **M**obile Communications
HPC	**H**igh-**P**erformance **C**omputing
HSPA	**H**igh **S**peed **P**acket **A**ccess
IFFT	**I**nverse **F**ast **F**ourier **T**ransform
IoT	**I**nternet **o**f **T**hings
IPC	**I**nter-**P**rocess **C**ommunication
ISS	**I**nstruction **S**et **S**imulator

ITRS	**I**nternational **T**echnology **R**oadmap for **S**emiconductors
LTE	**L**ong **T**erm **E**volution
LSB	**L**east **S**ignificant **b**it
MAC	**M**edia **A**ccess **C**ontrol
MCAPI	**M**ulti-core **C**ommunications **API**
MIMD	**M**ultiple **I**nstruction, **M**ultiple **D**ata
MPI	**M**essage **P**assing **I**nterface
MPSoC	**M**ultiprocessor **S**ystem-**o**n-**C**hip
MSB	**M**ost **S**ignificant **B**its
NI	**N**etwork **I**nterface
NoC	**N**etwork **o**n **C**hip
PE	**P**rocessing **E**ngine
PGAS	**P**artitioned **G**lobal **A**ddress **S**pace
POSIX	**P**ortable **O**perating **S**ystem **I**nterface for UNI**X**
QoS	**Q**uality **o**f **S**ervice
RDMA	**R**emote **D**irect **M**emory **A**ccess
RTT	**R**ound-**T**rip **T**ime
SCC	**S**ingle-chip **C**loud **C**omputer
SDF	**S**ynchronous **D**ata **F**low
SDR	**S**oftware **D**efined **R**adio
SIMD	**S**ingle **I**nstruction, **M**ultiple **D**ata
SMP	**S**ymmetric **M**ulti-**P**rocessors
SoC	**S**ystem-**o**n-**C**hip
TLM	**T**ransaction **L**evel **M**odeling
VLIW	**V**ery **L**ong **I**nstruction **W**ord

Introduction

The technology scaling predicted by Moore's law [1] continuously allows the integration of a higher number of components in a single chip, named SoCs (System-on-Chip). In the past few decades, along with increasing transistor count, the SoC performance could be improved by simply increasing its operating frequency. However, due to technology scaling limitations such as the power wall [2], this approach has reached its limit. Thus, in order to further increase embedded systems performance, architectures with multiple processing cores have been widely used in the past few years.

Nowadays, in addition to the higher performance requirement, multiple application fields (e.g., communication standards, high-quality video processing and computer vision) also impose low-power consumption as a primary constraint. Therefore, hardware accelerators might be employed to achieve higher power efficiency, i.e., higher performance with lower power consumption, since they are designed to execute specific tasks with a limited amount of resources, e.g., an FFT (Fast Fourier Transform) block, or a DSP (Digital Signal Processing) processor.

On the other hand, the addition of multiple processing cores coupled with dedicated hardware accelerators increases the system complexity and renders the application development a key challenge for two main reasons [3]. Firstly, embedded application workloads continue to grow in diverse fields (e.g. spatial, automotive, etc.) [4]. Secondly, the software technologies are not evolving as fast as hardware architectures, leaving a gap in the full system design [5]. In other words, custom software solutions that take benefit of given hardware platform aspects are required to decrease software overhead and fully explore each hardware architecture. Moreover, it force the users to build custom infrastructures to support their programming requirements, decreasing code compatibility, portability and reuse.

A solution to decrease this gap is the implementation of standard software APIs (Application Programming Interface), which is the approach adopted by several works presented in Section 1.2. However, these standards were developed in the context of SMP (Symmetric Multi-Processors) systems, and impose limitations regarding hardware

heterogeneity and memory footprint when addressing embedded systems. Indeed, the induced overhead might compromise the application performance and, consequently, violate the initial constraints. Hence, multi-core programming in embedded systems is hindered by the lack of flexible and general-purpose support.

Therefore, a flexible and lightweight standard specifically designed for embedded systems that addresses both homogeneous and heterogeneous architectures is desired. By using such solution, the software overhead could be decreased and applications could be efficiently ported between different architectures. A promising solution of software standards for embedded system is proposed by the Multi-Core Association (MCA), called Multi-core Communication API (MCAPI) [6]. This standard has been subject of many works, such as [5, 7–12]. Yet, as shown by some of these works, the application performance is also directly affected by the software stack/support efficiency, since MCAPI does not target a specific hardware architecture and a simple implementation will not take benefit of hardware aspects.

To counterpart this issue, the addition of hardware mechanisms able to handle and speed-up the inter-process communication can increase the overall system performance. On the other hand, most of the time, these mechanisms are not flexible to couple with different types of architectures and/or do not take into account the increased complexity in software development to manage them. Thus, it is mandatory to co-design hardware and software to increase the programmability of multi-core architectures and achieve the expected performance requirements. Moreover, the use of a standard software API is essential for code reuse and compatibility.

Motivation

This work is motivated by the need of co-designing hardware and software in order to increase applications programmability and efficiency while keeping good performance. The main idea of this approach is represented in Figure 1. The scenario illustrating a given application deployment in a heterogeneous hardware platform is depicted in Figure 1(a). In this scenario, the programmer needs to consider each processing core characteristics in order to code a given task accordingly. Although this approach allows the application to take advantage of hardware features, it compromises programmability, code reuse and portability.

On the other hand, Figure 1(b) illustrates that an API might increase the aforementioned aspects, allowing programmers, developers and users to abstract hardware

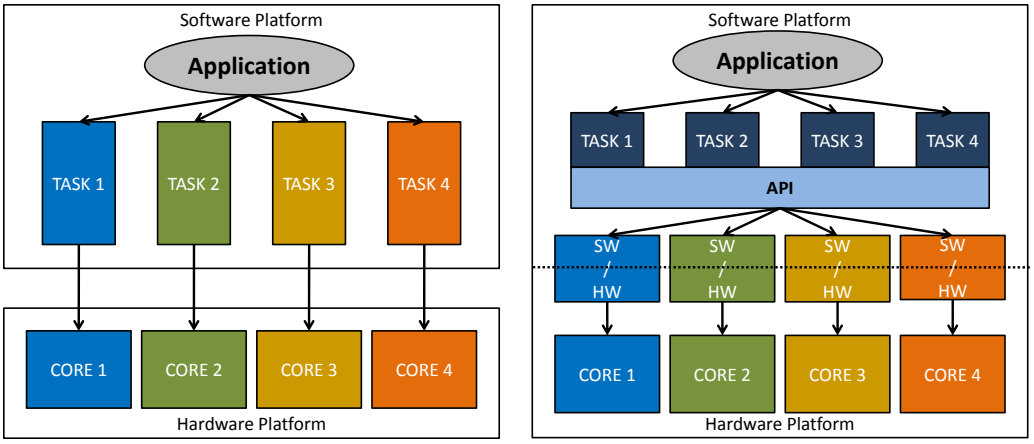


FIGURE 1: Application deployment in a multi-core architecture. (a) Without software support. (b) With software support.

details. Nonetheless, the application performance must be preserved, since the application requirements, e.g., output frame rate or data transfer rate, still have to be respected. Therefore, as both hardware and software support play an important role in the overall application performance, hardware or software mechanisms might be used by the API to extract the required performance leveraging on hardware features.

Therefore, this thesis focuses on developing the elements introduced in Figure 1(b). From the software side, a standard API targeting embedded systems is chosen, which allows to increase programmability and code portability/reuse at the same time. As the selected standard focuses on inter-process communication, the hardware mechanisms focus on increasing the communication performance. Furthermore, the different communication modes supported by the API are taken into account in order to co-design the mechanisms accordingly. In summary, the main objective is to propose hardware mechanisms that can alleviate the overheads imposed by the software implementation and increase communication performance.

Contributions

In order to achieve the main objective, the developments performed during this thesis concern platform software and hardware levels, as highlighted by the red line in Figure 2. At the platform software level, the APIs and HAL (Hardware Abstraction Layer) must be developed to couple with the reference architecture and increase software programmability. Next, in order to increase application performance without increasing software

complexity, the addition of modules or mechanisms in the platform hardware level must take into account the software level. Thus, three main contributions are presented:

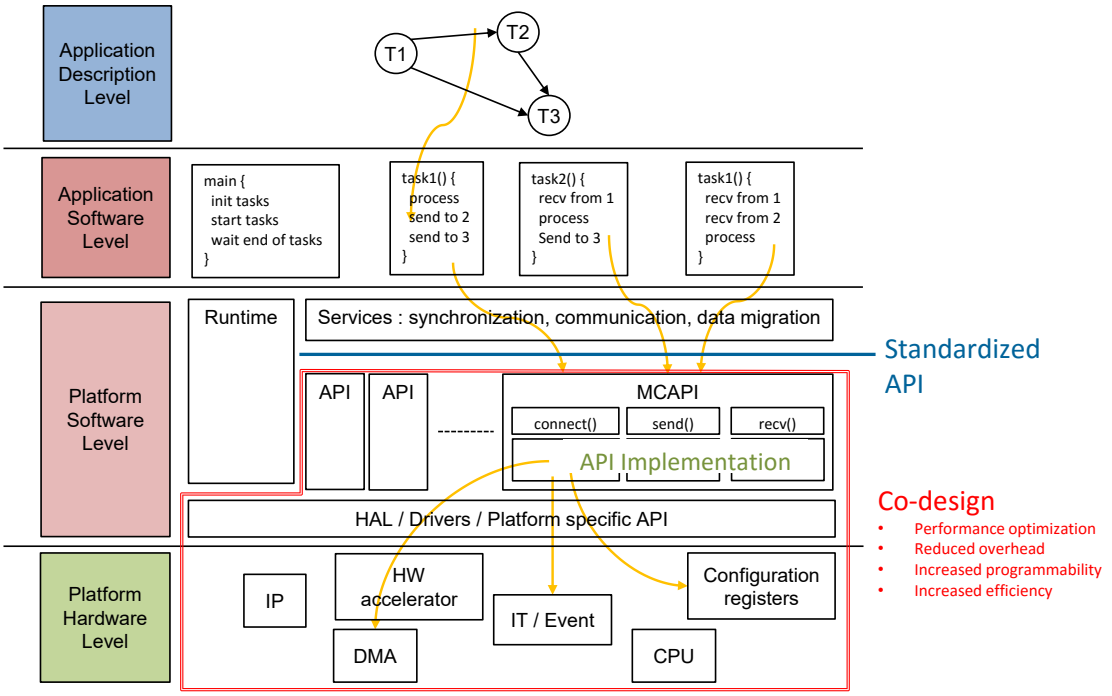


FIGURE 2: Co-design approach regarding different abstraction levels.

- **Implementation and evaluation of a standard API for inter-process communication**.
  MCAPI allows the applications to express inter-process communication in a standardized manner, increasing code portability and reuse. Therefore, the communication can be implemented independently from the hardware architecture. Additionally, we characterized the overhead imposed by the software implementation, showing which points must be improved in order to avoid performance losses.

- **Design and evaluation of a hardware mechanism to improve communication set-up performance**.
  We developed a hardware mechanism in co-design with MCAPI in order to decrease network traffic and processor workload during the communication set-up phase. Due to the co-design approach, this mechanism can be easily accessed and programmed at the API level, making the hardware modifications transparent to the application.

- **Design and evaluation of a hardware mechanism to improve data transfer performance**.

We co-designed a hardware mechanism with MCAPI, which implements a FIFO-like communication channel and handles the data transfer phase. This mechanism is programmed through memory-mapped registers, making it flexible to be used by processors or hardware accelerators. Similarly to the communication set-up mechanism, the software modifications are performed only at API level, without increasing programming complexity.

Thesis Organization

The thesis is divided into Introduction, five main chapters, and Conclusion. The Introduction describes the context of this work and presents its motivation and contributions. Chapter 1 places the thesis in relation with the state-of-the-art and provides background in multi-core architectures and software programming interfaces. Since this work is inserted in the context of multi-core architectures programmability, the hardware-software co-design aspect is also discussed, as well as hardware mechanisms looking for performance improvements. Additionally, it presents the reference architecture used to develop the contributions of this thesis.

Chapter 2 presents the MCAPI standard and the design choices performed to implement the specification for the reference architecture. Furthermore, an analysis is carried out to evaluate the main drawbacks and bottlenecks in the referred implementation.

Then, Chapter 3 presents the Event Synchronizer mechanism, which is the second contribution of this thesis. It also presents the polling processes present in the MCAPI implementation and the modifications performed in the MCAPI implementation to take advantage of the Event Synchronizer. Chapter 4 presents the third contribution of this thesis, which is the Buffer Manager Mechanism. Initially, the data transfer phase and the software implementation are reviewed. Then, the proposed mechanism is described. Furthermore, the modifications performed in the MCAPI implementation to take advantage of the Buffer Manager are presented in the end of the chapter.

The environment set-up, experimental results and benchmark validation are described in Chapter 5. The objective is to characterize the mechanisms proposed in Chapters 3 and 4, as well as evaluate the MCAPI implementation described in Chapter 2 in terms of memory footprint. Furthermore, the performance gains obtained with the proposed mechanisms are validated through the execution of image processing and path calculation benchmarks. Finally, conclusion and perspectives are drawn.

Chapter 1

Multi-core Systems Overview: Hardware and Software Related Works

This chapter places the thesis in relation with the state-of-the-art and provides background for multi-core architectures and software programming interfaces. Since this work is inserted in the context of multi-core architectures programmability, the hardware-software co-design aspect is also discussed. Therefore, Section 1.1 presents examples of recent multi-core platforms, providing insights to architecture trends. Then, Section 1.2 gives multiple options of software support that can be applied on these architectures. Section 1.3 introduces works related to hardware mechanisms looking for improving the performance of such architectures and discuss their support or interaction regarding the software level. Finally, the reference architecture used to develop the contributions of this thesis is presented in Section 1.4.

1.1 Multi-core Architectures

Multi-core architectures have been used in the past years as solution to meet application constraints in several areas, such as high-bandwidth digital communication, gaming, augmented reality and high-quality image and video encoding. In addition, according to ITRS [13], the number of processing engines in a single device is expected to grow exponentially in the next years, as depicted in Figure 1.1, which confirms the importance of further developing such architectures.

Besides the processing engine count, the core types must also be considered. When more than one core type is employed, the multi-core platform can be classified as heterogeneous. On the other hand, homogeneous architectures are composed of a single replicated processing engine, which can be either general purpose processors (GPPs) (e.g., Intel's multi-core CPUs, SCC [14]) or DSP processors [15]. Both homogeneous and heterogeneous architectures have their advantages and drawbacks. While applications are easier to code and parallelize in homogeneous architectures, the power efficiency of heterogeneous solutions is higher due to the specialized hardware units employed to execute specific tasks.

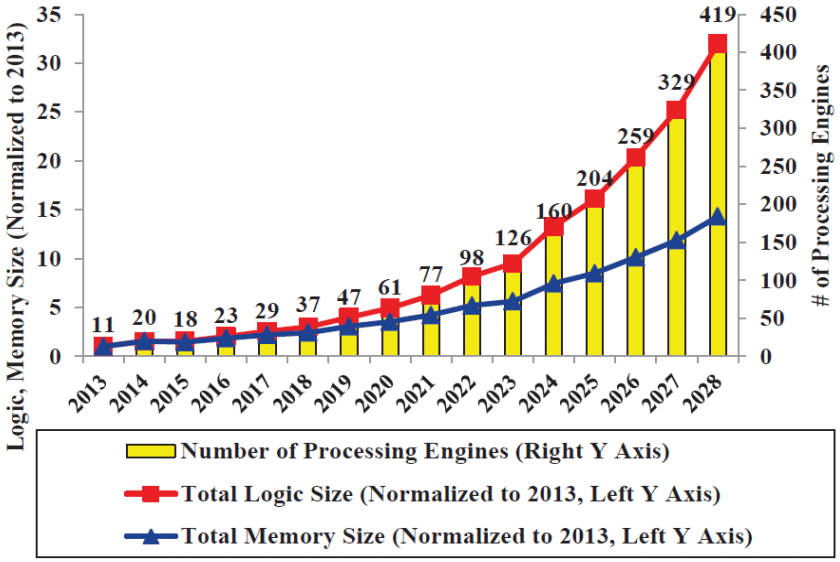


FIGURE 1.1: Design complexity forecast for SoCs in terms of processing engines, logic and memory sizes [13].

In fact, heterogeneous solutions are being widely employed in both industry and academy. Additionally, according to ITRS, the SoC template consists of few processing cores (4 cores, currently), GPUs (Graphics Processing Unit) and several processing engines (PEs) to execute specific tasks (Figure 1.2), which characterizes a heterogeneous architecture. Furthermore, "due to the continuously increasing demand for high-definition audio and video playback, the number of GPUs is expected to rapidly increase" [13]. Therefore, it is safe to affirm that heterogeneous architectures will be preferred over homogeneous architectures due to the increased performance and higher power efficiency.

The industry has already demonstrated many examples of architectures composed of GPPs and GPUs, such as Tegra from NVIDIA, Exynos from Samsung, Ax series from Apple, Fusion APU from AMD and Ivy, Sandy and Haswell architectures from Intel.

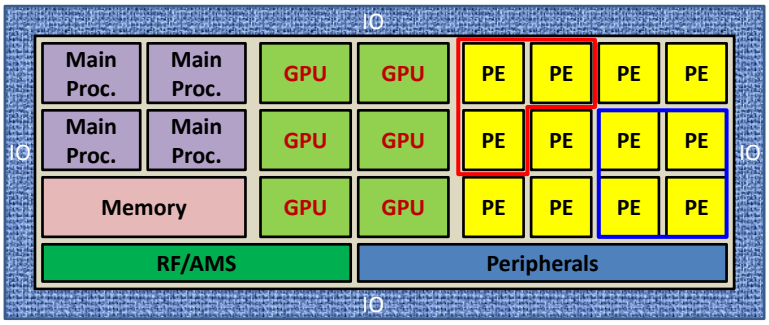


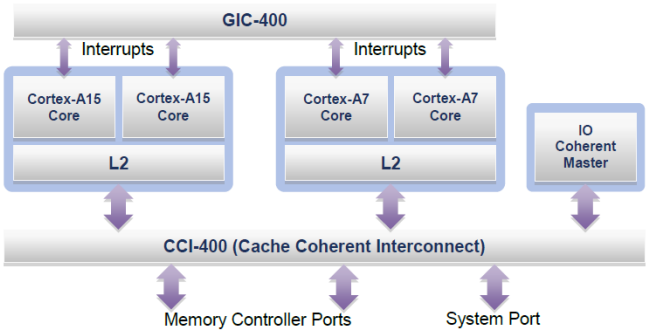
FIGURE 1.2: SoC architecture template for consumer portable devices [13].

Figure 1.3 details the block diagram of two industry architectures: big.LITTLE [16] and Cell Processor [17], from ARM and STI, respectively.

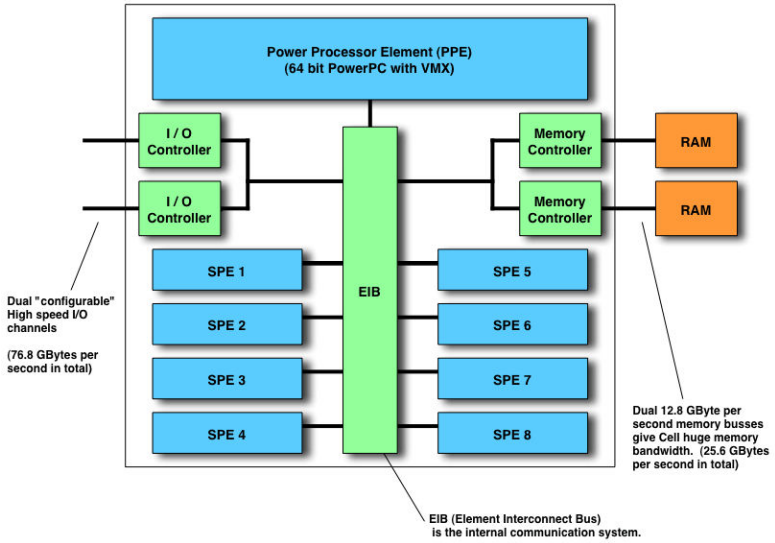
Figure 1.3(a) shows the big.LITTLE block diagram, which consists of an ARM Cortex-A15 pair (big) and an ARM Cortex-A7 pair (LITTLE) connected by the cache coherent ARM CoreLink CGI-400 interconnect. The idea is to take advantage of different workload patterns in mobile applications to switch the operation between the different cores. The "big" cores are used when high processing power is needed, while the "LITTLE" cores are employed to execute simpler tasks. This architecture is used in the Exynos 5 [18] and 7 Octa from Samsung and in Qualcomm Snapdragon 808 and 810 processors.

The Cell processor block diagram is presented in Figure 1.3(b). It is composed of one Power Processor Element (PPE) and eight Synergistic Processing Elements (SPEs). The interconnection is performed by the Element Interconnect Bus (EIB), which is a set of 4 ring buses. The SPEs are responsible to execute computing intensive tasks assigned by the PPE, which can also run an operating system and applications. This architecture is used in PlayStation 3 gaming console and was developed in co-operation between Sony, Toshiba and IBM [17].

Heterogeneous multi-core architectures are also research subject in the academy, and several examples can be highlighted. The Tomahawk MPSoC [19] targets baseband communications and multimedia applications. The platform is composed by two Tensilica processors (DC212), six fixed-point DSP units (VDSP), two scalar floating-point DSP units (SDSP), among other specialized hardware blocks, as depicted in Figure 1.4. All components are connected by two crossbar-like master/slave NoCs. The application control and scheduling is centralized in the CoreManager unit. The platform is programmed using a C-based programming model. The programmer explicitly assigns the C-functions that will be executed as tasks in a given PE using special *#pragma* directives. Before task execution, program and input memories are copied from the global



(a) big.LITTLE block diagram overview [16].



(b) CELL processor architecture [17].

FIGURE 1.3: Block diagrams for biggle.LITTLE and CELL processors.

to the local PE memory. After task completion, the local PE memory is copied back to the global memory.

There are also architectures targeting only a single application field, such as Magali [20], X-GOLD SDR 20 [21] and COBRA [22]. These architectures are focused mainly on Software Defined Radio (SDR) applications. Magali [20] is an evolution of the Faust [23] platform, which is composed of 23 processing units, including VLIW DSP processors, blocks dedicated for FFT/IFFT processing, a hardware block for managing configuration and data (DCM) and an ARM processor working as a centralized general manager. These units are connected by a 3x5 mesh asynchronous NoC (ANoC), as presented in Figure 1.5. Also, this architecture supports dynamic data-flow reconfiguration. Using a data-flow programming model, the ARM processor configures the Communication and Configuration Controllers (CCCs), which are in charge to manage the communication in a distributed way. The data flow is statically defined at compile time, but the number of data to be exchanged between the different units can be modified dynamically. When

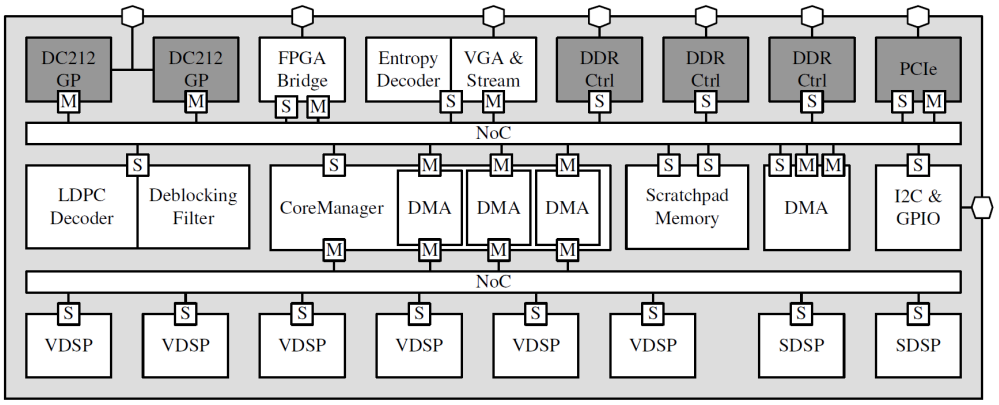


FIGURE 1.4: Tomahawk block diagram overview [19].

a new configuration of a given task is missing, the CCC requests a configuration to one of the DCMs, which stores all the configurations defined at compile time.

The X-GOLD SDR20 [21] platform supports the implementation of the physical layer signal processing for GSM, EDGE, HSPA, LTE and others. The processor contains an SDR subsystem, an ARM subsystem and a power management unit. The ARM subsystem is composed by the ARM 1176 core, local memories, a DMA, external memory interface, an audio DSP processor, among other peripherals. The SDR subsystem is composed by SIMD clusters and an accelerator cluster. The SIMD cluster is composed by four SIMD cores, two RC1632 scalar cores, a shared memory, a multi-layer local bus, and a bridge to connect the cluster to the Global Bus. The scalar cores are intended to be used for tasks without parallelism. Similarly, the Accelerator cluster contains five accelerators in total. Each accelerator is composed of a RC1632 control processor and the actual accelerator core, plus a shared memory. In the same way of the SIMD

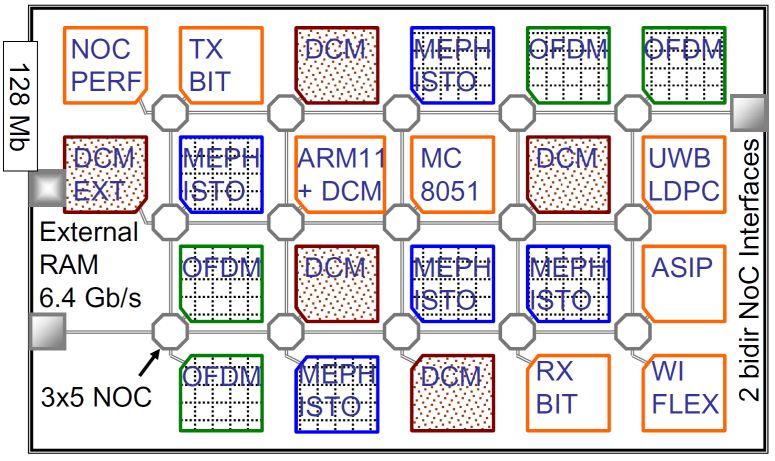


FIGURE 1.5: General overview of Magali processing cores [20].

cluster, the interconnection is done via a multi-layer local bus. Additionally to the clusters and ARM subsystem, a dedicated mechanism allows synchronization between tasks. However, there is no mention about the programming model or tool chain to assist the programmer developing applications for this platform.

Lastly, the COBRA (Cognitive Baseband Radio) platform [22] is a multi-core SDR platform based on the ADRES baseband processor [24]. This processor consists of one or more VLIW processors and a set of Coarse Grain Array (CGA) resources. Each VLIW processor is able to run the application sequential code, while the CGA runs the parallel code. The COBRA platform contains 2 ADRES baseband processors, 2 DIFFS (Digital Front For Sensing), 2 flexible forward error correction (FlexFEC), a Viterbi accelerator and an ARM core for task controlling. All parts are programmable in C or high level assembly. The interconnection is done via buses and crossbars. From the memory hierarchy point of view, there are local memories, which are connected to each thread and store local variables, and a shared memory, which stores shared variables that are used by different threads or implement FIFO communication between threads. The application are implemented using the MPA tool [25], which uses the application C code and a file specifying the parallelization as input. Then, the tool analyzes the dependencies in the C code, performs the parallelization and inserts FIFO buffers at the points where communication between threads is needed.

However, more recent architectures are focused on flexibility and distributed control, such as Flextiles [26] and P2012 [27]. The FlexTiles platform targets high performance and dynamic algorithms in embedded products with low power consumption. The architecture is scalable, and there is not a fixed number of cores that the platform supports. Indeed, it is composed by general purpose processing (GPP) nodes, DSP processors, Accelerators Interfaces (AIs), an embedded FPGA, I/Os and a DDR controller. The GPPs are used as masters while the DSP processors, eFPGA and I/Os are used as slaves.

The interconnection infrastructure employs a NoC for communication. The interface between NoC and GPP nodes, DDR controller, I/Os and eFPGA Configuration Controller is done via a Network Interface (NI). In addition, for the DSP nodes, an AI is used between the DSP core and the NI. The platform is not yet implemented in silicon. However, for future implementations, it is proposed the use of 3D stacking technology. In this case, eFPGA module is placed in a different layer from the GPP nodes.

The programmer's view is a set of concurrent threads with different priorities that can address domain-specific accelerators to meet application constraints. The application is programmed in C or C++ language, while the accelerated parts are described in

the respective accelerator language. The application parallelism is expressed by describing the threads as series of data-flow graphs and combining them with priority orders and synchronization mechanisms. In fact, the application is defined as a set of static clusters. Then, a cluster is described using Synchronous Data Flow (SDF) or Cyclo-Static Data Flow (CSDF) models of computation, where each producer/consumer is defined as an actor.

Finally, Platform 2012 [27] is a multi-core architecture composed of processor clusters in a GALS approach targeting applications such as digital communication standards, gaming, augmented reality, high-quality audio and video, etc. Each cluster is composed of a computer engine called ENCore, which can be composed from 1 to 16 PEs (STxP70-V4), a cluster controller and several Hardware Accelerators (HWPEs). The cluster controller is responsible for booting and initializing the ENCore PE, application deployment, error handling and also energy management. In fact, each cluster has his own cluster controller, characterizing a distributed controlling approach. The ANoC performs communication intra (between HWPEs and ENCore engine) and inter-cluster.

The platform supports three classes of programming models: (i) Native Programming Layer (NPL); (ii) Standards-based Programming Models; (iii) Advanced Programming Models. Using (i) the platform resources are accessed through a low-level C-based API. In (ii) some standard programming models can be used to program the platform, such as OpenCL and OpenMP. Finally, (iii) offer a midway productivity/performance trade-off between the other two. From the software viewpoint, the 2012 architecture is PGAS – Partitioned Global Address Space, where all processors have full visibility on all the memories with no aliasing, i.e., a processor located in a given cluster can load and store directly in remote L1 memory of other clusters.

Table 1.1 compares the mentioned architectures regarding their target applications (specific, semi-specific or generic), inter-process communication (IPC), communication control, hardware characteristics (interconnection and processing engines) and support for application programming.

The architectures' target applications differ and a trend can not be identified. However, with mobile devices supporting multiple features (4G and 5G standards, high-quality video playback, etc) and Internet of Things (IoT) becoming increasingly in evidence, architectures targeting specific application niches are not suitable. The predominant inter-process communication implementation is through shared memories with centralized communication control. On the other hand, architectures supporting a high number of processing nodes must implement distributed control to avoid a bottleneck in the system. Moreover, only P2012 supports standard programming models. Therefore,

TABLE 1.1: Multi-core architectures summary and comparison.

Architecture	**Target Application**	**IPC**	**Communication Control**	**Interconnection Infrastructure**	**Main Processing Engine**	**Hardware Accelerators**	**Application Programming**
big.LITTLE [16]	Generic	Shared Memory	Centralized	Proprietary Interconnect	ARM-based processors	-	-
Cell [17]	Generic	Shared Memory	Centralized	Ring buses	Power PC	Synergistic Processing Units	-
Tomahawk [19]	Semi-Specific	Shared Memory	Centralized	NoC-based	Tensilica DC212	VDSPs, SDSPs, LDPC Decoder, etc	C-based; #pragmas
Faust [23]/Magali [20]	Specific	Message Passing	Partially Distributed	NoC-based	ARM-based processor	FFT, IFFT, VLIW DSPs, etc	Data-flow configuration
X-GOLD SDR20 [21]	Specific	Shared Memory	Centralized	Bus-based	ARM-based processor	SIMD processors, Audio DSP, etc	-
COBRA [22]	Specific	Shared Memory	Centralized	Crossbars and buses	ADRES	DIFFS, FlexFEC, Viterbi accelerator	MPA tool
Flextiles [26]	Generic	Shared Memory and Message Passing	Distributed	NoC-based	GPPs	DSPs, eFPGA	Thread-based; specific languages
P2012 [27]	Generic	Shared Memory	Partially Distributed	NoC-based	STxP70-V4	HWPEs	Multiple options

the multi-core architecture presented in Section 1.4 is used as reference, taking flexibility and scalability as the main target constraints.

1.2 Application Programming Interfaces

Besides the hardware architecture, efficient implementation of the software stack is also important for two main aspects: application performance and portability/code reuse. Usually, the application take advantage of Application Programming Interfaces (APIs) or another software component that interfaces with the hardware. Parallel and distributed computing have well established software standards for both shared memory and message passing paradigms. On the other hand, the lack of standards for embedded systems often leads to custom solutions.

1.2.1 Custom APIs

Several works present implementations of lightweight solutions for well-known standard APIs when targeting embedded systems. In [28], only a set of MPI ([29]) functions are implemented for four different network topologies. The code size is reported to be between 11 kB and 16 kB. Similarly, [30] and [31] present optimizations for a set of MPI functions. However, since only a set of MPI functions is implemented, application portability might be compromised. Other works present solutions with specific architecture constraints, such as [32], where a processor with enough processing power to run a full MPI implementation is needed, or such as [33], where the solution targets CPU-GPU architectures. Even though these implementations take advantage of MPI, they can not be considered a standard. Furthermore, [34] and [35] report performance overheads induced by software implementation that might compromise the application constraints.

Moreover, other custom software solutions can be highlighted. C-Heap [36] (CPU-controlled Heterogeneous Architectures for Signal Processing) is a combination of an architecture template with a protocol for cooperation and communication between autonomous tasks. The architecture template is a MIMD (Multiple Instruction Multiple Data) architecture, and has a main processor controlling the other processing devices.

The communication protocol is implemented using FIFO-based communication between the tasks. In this model, the communication is divided between synchronization and data transportation. As the FIFOs are implemented using shared memories, no data copy is necessary and only synchronization primitives are needed. In this architecture,

the data transfers are based on tokens. Therefore, the tasks need to claim and release tokens to manage the buffer memory space using primitives. These primitives, used by the tasks, control the allocated space in a FIFO manner to implement a channel FIFO buffer. Therefore, it provides flexibility to tune the FIFO and token sizes for an application even after a silicon tape-out. Also, the number of FIFOs and their interconnection structure can be changed in order to map different applications on the same hardware.

In [37], a message passing library with limited number of functions is presented. The focus is to decrease the overhead imposed when setting up and controlling the communication and in the data transfer. The library uses semaphores for data synchronization, which are explicitly informed in the functions calls. The synchronization can be done by polling or by waiting for an interruption for a given semaphore. Although performance gains are shown, the library provides limited functionality and is not based on a widely used standard.

A different approach is proposed in [38], where a programming interface to express communication constraints at language level is presented. The constraints are evaluated by the operating system, which configures the communication hardware accordingly. The base programming language is the X10, which offers type safety, system modularity, portioned global address space, dependent types, transactional memory, among other features. The optimizations in the communication are done by resource allocation and data pre-fetching through DMA.

Nevertheless, the main issue related to these solutions are code portability and reuse, since the API, software framework or programming language are not widely employed. Therefore, as the multi-core hardware is evolving faster than software technologies, other software standards are necessary to address embedded systems capabilities [5].

1.2.2 Standard APIs

Implementing a standard software solution provides benefits such as code portability and code reuse. An example of a well-know standard is the OpenMP API [39]. It supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran. In a multi-thread implementation, the master thread forks a specified number of slave threads to work over a given set of data. Then, each slave thread executes its parallelized code section. After the slaves processing completion, the slave threads join back into the master thread, which continues to execute until the end of the program or until the next fork. Thus, the work-sharing constructs can be used to divide a task among threads so that each one executes its allocated part of the code.

The code section that will be executed in parallel is defined by a preprocessor directive, e.g. #pragmas in C language. The number of threads that will be generated can be defined by the user, or left to be managed by the runtime environment. In the later case, the runtime environment allocates threads to processors depending on usage, machine load and other factors.

Another standard is the MPI (message passing interface) [29], which is a message passing library standard for parallel programming. Therefore, MPI is not a library itself, but the specification for developers and users of what this library should be. The goal of the MPI is to provide a widely used standard for writing message passing programs. The interface specifications are defined for C/C++ and Fortran.

The main advantages of using MPI includes the portability, since there is no need to modify the source code when porting an application to a different platform that supports the MPI standard, and functionality, with more than 115 routines implemented already in MPI-1. Also, implementations are available at both vendor and public domain [40].

Initially, MPI was designed to be used in distributed memory architectures. However, as the architecture trends changed, shared memory SMPs (Symmetric Multi-Processors) were combined over networks, creating hybrid distributed/shared memory systems. Then, the MPI implementors adapted the libraries to handle both types of underlying memory architectures. Nowadays, MPI can run virtually on any kind of memory architecture (distributed, shared or hybrid), but the programming model remains as a distributed memory model. As a drawback, all parallelism is explicit, i.e. the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using the MPI constructs.

More recently, languages such as OpenCL [41] and OpenCV [42] were developed to program heterogeneous multi-core platforms. OpenCL is an open standard maintained by the technology consortium Khronos Group. It has been adopted by Intel, AMD, NVIDIA, Altera, Samsung, ARM, among others. The OpenCL framework is used for writing programs that execute across heterogeneous platforms consisting of CPUs, GPUs, DSP units and other processors. Also, it includes a language (based on C99) for writing kernels, and APIs that are used to define and then control the platforms.

The programming language used to write computation kernels is based on C99 with some limitations and additions. It omits the use of function pointers, recursion, bit fields, variable-length arrays, and standard C99 header files, but provides some extensions. The platform model considers one host device and one or more compute devices. In turn, each compute device is composed of one or more compute units that can be divided processing elements. The applications written in OpenCL are managed by the host, which submits

the work to compute devices. The memory management is explicit, which means that the programmer must move the data manually from/to any memory level, e.g. from host memory to local memory, or vice-versa.

OpenCV s an open-source library including several hundreds of computer vision algorithms that can take advantage of multi-core processing. It was firstly developed by Intel, and now is supported by Itseez. The main focus is on real-time image processing, but the functions also cover several areas such as machine learning, camera calibration, stereo, 3D, among others. The library is written in C++, but there are interfaces in Phyton, Java and Matlab.

Table 1.2 compares the aforementioned standards and also MCAPI, which is introduced in the next section. Although multiple standards are available to implement parallel applications, embedded systems with limited memory and power budgets are not addressed. Indeed, the OpenCL standard does target embedded systems, but it imposes a model between host and computing devices that might not fit in some platforms. In the same way, OpenMP API focus only on shared memory architectures and benefits mainly SMP (Symmetric Multiprocessing) machines, lacking support for heterogeneity. On the other hand, MPI and OpenCV could be explored by embedded systems. However, OpenCV needs to be executed on top of an operating system or other run-time environment, which prohibits its use for platforms with limited resources. Similarly, the implementation of the entire MPI standard is too heavyweight for such architectures [5].

TABLE 1.2: Comparison of different software standards for multi-core programming.

API/Library	**Target Architecture**	**Target Application**	**Language**
OpenMP	Shared Memory based	Fork-join algorithms	C, C++, Fortran
MPI	Any	Generic	Any
OpenCL	Heterogenous	Generic	OpenCL, C99
OpenCV	Any	Computer Vision and Machine Learning	C++ and interfaces for other languages
MCAPI	Any	Generic	Any

1.2.3 MCAPI

The MCAPI (Multicore Communications API) specification [6] was created by the Multicore Association, an association with several industry companies, and defines an API and semantics for communication and synchronization between processing cores in embedded systems. This API uses the message passing communication model. The purpose is to capture the basic elements of communication and synchronization that are required for closely distributed embedded systems. The primary goals of MCAPI implementations are extremely high performance and low memory footprint. Additionally, MCAPI

intends to be scalable and to virtually support any number of cores, each with a different processing architecture and running the same, different, or no operating system at all. Thus, MCAPI aims to provide source-code compatibility that allows applications to be ported from one operating environment to another.

Three communication modes are provided: messages, packets, and scalars. Also, MCAPI provides a limited number of calls with sufficient communication functionality while keeping it simple enough to allow efficient implementations. In comparison with other communication APIs such as MPI, the MCAPI targets inter-core communication in a multi-core chip, while MPI was developed targeting inter-computer communication. Furthermore, other characteristics differ MCAPI from MPI, such as the support of quality of service and priorities, the absence of collective communications, no concept of groups, no synchronization methods (barriers, fences, locks), among others. Similarly, in comparison with OpenCL and OpenCV, which more closely resembles a programming model, MCAPI does not focus on architecture or application type, but in the inter-process communication. Nevertheless, both standards can take benefit of MCAPI to implement their low layers communication protocols.

Although the standard does not entirely cover multi-core programming, it provides a solution for the communication aspect and capability to implement a significant amount of applications. Additionally, multiple implementation examples are already available. For example, the MCAPI implementation presented in [5] is publicly available and can run on Linux computers. It separates the implementation in 2 layers: common code with MCAPI function calls as presented in the specification, and a transport layer, implementing the MCAPI functionalities through Posix shared memory and semaphores. Another example of implementation using shared memory was performed by Mentor Graphics [7]. In [8], the standard was extended to allow inter-chip communication. Finally, works presenting MCAPI implementations for FPGA [9–11] and Intel's SCC [12] are also available.

Thus, the MCAPI standard is a relevant development choice in the context of this thesis. As mentioned, the MCAPI is an increasingly popular standard, which allows the implementation described in Chapter 2 to be compared with other solutions, which is performed in Section 5.2. In addition, the MCAPI main characteristics, i.e., focus on embedded systems, low memory footprint, high communication performance, scalability and heterogeneity, are fully compatible with the objectives of this thesis.

1.3 Communication Hardware Mechanisms

Improving communication performance has been research subject for many years. Initially, the main concern was to decouple communication and computation in clusters of processors through hardware support for inter-process communication [43]. However, as showed in [44], both hardware support and software programming model must be addressed in order to successfully implement efficient communication. Yet, the Authors in [44] propose a custom programming model as solution. Similarly, [45] presents software optimizations to increase communication performance in SMP clusters by decreasing synchronization overhead.

Nonetheless, the research on hardware/software co-design for embedded systems must present different solutions, since its characteristics greatly differ from SMPs. In addition, multi-core architectures present high programmability complexity due to limited software capabilities, e.g., smaller operating systems to couple with limited memory resources. Usually, increasing programmability induces performance overheads [10–12, 46]. Thus, it is mandatory to co-design hardware and software to increase programmability while meeting application constraints in multi-core architectures. Although aiming a specific SoC and proposing a software custom solution, an example of hardware/software co-design is presented in [47] and shows significant performance gains.

Since MCAPI targets inter-process communication, the research must focus on solutions that improve this aspect. Considering the three communication modes, messages is the most simple, since the data can be transferred by informing only the endpoint identifiers. However, for packet and scalar channels, specific actions must be executed by both communication endpoints before and after exchanging data. Indeed, for these modes, the communication process can be split in two phases: communication set-up and data transfer, as depicted in Figure 1.6. The first phase is used to establish a connection, allocate and deallocate resources, while the second one is where the data is actually exchanged. Therefore, both phases must be taken into account when developing the hardware mechanisms. The following sections describe related works for synchronization mechanisms (that could be used during communication set-up) and data transfer support. In the end, the work presented in this thesis is compared to the works described in both sections.

1.3.1 Synchronization/Communication Set-Up

The communication set-up can be considered a set of synchronization steps, since the MCAPI standard defines conditions that can be interpreted as a barrier, or a lock

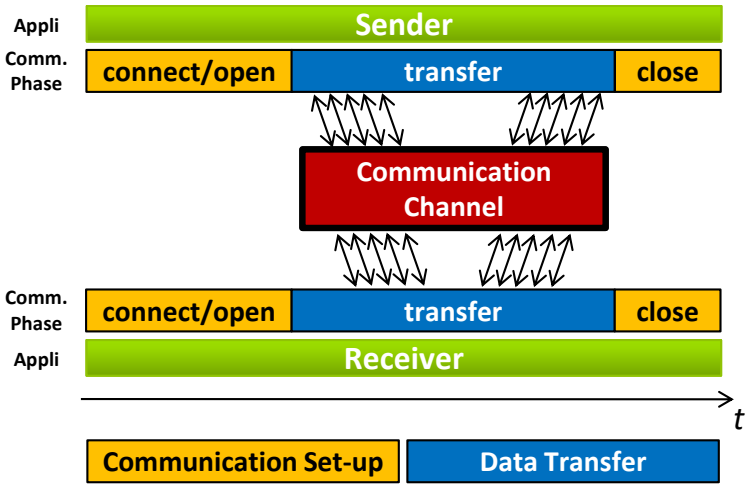


FIGURE 1.6: Communication set-up steps used by packet and scalar channels.

in shared resource. According to [48], an ideal synchronization mechanism has to be flexible, scalable, contention free and present low latency and low overhead. Based on these characteristics, a mechanism focused on thread synchronization is proposed in [49]. The idea is to take advantage of the "*weak synchronization principle*" and tightly-coupled accelerators to achieve high core utilizations. Although the mechanism is easily usable by the applications, its flexibility is limited. This limitation is related to the fact the mechanism only solves the issue of scheduling synchronization for multi-core architectures employing tightly-coupled cores.

A more flexible solution is presented in [50], where the scratch-pad memory is used to virtualize synchronization and communication operations. This way, the software layer can issue operations by writing in the virtual addresses. The synchronization primitives are provided by counters and queues. The counters provide software notifications upon completion of a given number of synchronization operations, e.g. a barrier, in the pre-configured address. Queues are used to send and received data, but can also be used to implement a lock or a semaphore. The main drawback of this solution is related to the architecture constraints, since communication operations are expected to be handled by a remote DMA (RDMA) due to address translation when accessing remote memories. Moreover, the software support for the proposed mechanisms is not mentioned.

Optimizations aiming the MPI standard in heterogeneous SMPs clusters are proposed in [51]. In this work, barrier operations are optimized automatically according to architecture profile. This is achieved by selecting different barrier algorithms according to profiles extracted for different system topologies. This solution provides higher

flexibility in relation to the target architecture and performance. However, implementing this solution in embedded systems might induce significant processing and memory footprint overheads.

Synchronization mechanisms targeting embedded systems are presented in [52–55]. A flexible mechanism to improve synchronization primitives in the P2012 [27] architecture is proposed in [52]. The mechanisms is composed of atomic counters and a programmable notifier hardware. The processing cores use memory-mapped registers to program the synchronization operations and, if a notification must be generated, interruption lines are used to notify the cores. In addition, the characterization of memory accesses to perform several synchronization primitives is detailed. However, the software implementation is mentioned only for a non-standard API, called RTM. Furthermore, as all the cores use the same module for synchronization, the scalability might be compromised.

The SCC [14] provides a hardware module called Messaging Passing Buffer (MPB), which is a software controlled memory of 16 kBytes that can be used for communication and synchronization. In [53], barrier algorithms are used to improve one-sided communication by taking advantage of the available hardware. The modifications are integrated in a MPI custom solution for the target architecture.

In [54] a synchronization method called *C-Lock* is presented. It aims to gather the advantages of both lock-based and lock-free methods. While lock-based methods require simpler control, lock-free methods access shared data in an optimistic manner, considering that conflicts will not occur. However, if a conflict does occur, a rollback and/or re-execution must be performed, generating processing and power overheads. Thus, the proposed mechanism targets to detect true data conflicts by considering the type, address range, and dependency of simultaneous memory accesses. As result, rollbacks are avoided and the cores with identified conflicts have their clock signals gated, increasing energy efficiency. However, this mechanism addresses only shared memory systems where mutual exclusion is needed to access the data. Additionally, the application code must be changed in order to take advantage of the hardware mechanism.

In fact, when dealing with architectures composed of a high number of processing cores, the lock-based method should be avoided due to low efficiency, as mentioned in [55]. The authors evaluate different message passing algorithms for shared data synchronization from HPC (High-Performance Computing) domain. The evaluations are performed in a platform that provides hardware support for basic synchronization primitives. The results show that synchronization through message passing can be also efficient in embedded multi-core architectures.

The comparison between these works and the thesis is performed in Table 1.3 (page 26).

1.3.2 Data Transfer

The data transfer phase occurs after performing the communication set-up. Indeed, data transfer performance in multi-core architectures is subject of many works, since it directly impacts the overall application performance. Most of these works present solutions to improve data transfer performance by implementing hardware mechanisms that perform operations usually handled by software, or by improving a given point of the target architecture. Architectures and mechanisms addressing the two most common programming models (shared memory and message passing) are considered.

A hardware mechanism for distributed memory architectures is proposed in [56]. The mechanism is composed of several components that handle the communication of the processing nodes (Memory Sever Access Points – MSAP), a control network and a data network. Each MSAP has several input and output ports connected to dedicated FIFOs for data transfers. The connection is created by linking an input to an output port. Though the results show the scalability of the proposed mechanism and a performance increase for the evaluated applications, there is no mention about software support or how the application is able to benefit from the hardware mechanism. Moreover, it barely differs from a DMA with an arbiter, where the processing element generates the requests and the arbiter configures the ports for data transfers.

Buono et. al. [57] presents a delegation mechanism that performs the communication between a producer and a consumer. This mechanism works in a DMA fashion. It can be implemented by SW (thread) or hardware. The process/task is responsible for passing a delegation descriptor that indicates the channel identifier and the message pointer. Then, the mechanism is in charge of sending the message through a buffer available from a static array. An application of this mechanism is channel creation in MCAPI. However, there must be one mechanism for each endpoint, which may generate significant overheads. The software support is implemented by a simplified custom solution.

In [58], the message-passing operations are handled in hardware. The proposed mechanism provides point-to-point DMA communication and collective communication operations (barrier, broadcast, collect, etc). Implementation details are not provided and it is not presented how the hardware mechanisms are programmed neither the used programming API. Similarly, [59] addresses the message passing performance by proposing a hardware mechanism based on a DMA engine. The mechanism offers the

possibility to program transfers by informing task IDs or memory addresses, increasing the flexibility. There are buffers to store send and receive operations. The software support is provided by a set of custom functions to send and receive data.

Collective communication operations are also addressed in [60]. The objective is to offload the software processing for broadcast, scatter, gather and reduction collective communications through a hardware module interfacing the processing core and network interface, called Smart Network Adapter. The mechanism is based in the MCAPI definitions, where data is transmitted between a pair of endpoints. However, the current MCAPI specification release does not provide collective operations. Furthermore, the communication channels are not taken into account.

A mechanism for communication and control of data-flow applications is introduced in [61]. It aims to distribute the data-flow configuration between the resources in the NoC and to increase the programming flexibility of such applications. Firstly, a boot step is performed by the host, which sets initialization parameters (e.g., NoC topology, global identifiers) and communication configurations to the configuration servers. Then, the configuration servers are used during the application execution by the tasks when a new communication configuration is needed.

Contrary to [59], the flow control implemented in this work is performed by credits, which also allows task synchronization, i.e., end of credits means that the task communication has finished. The mechanism can be programmed through a custom instruction set, allowing different configurations to be loaded without changing the hardware. However, to decrease the mechanism complexity, this instruction set is restricted to few instructions, aiming only applications with deterministic scenarios and without data-dependency.

In order to counterpart this issue, a mechanism to support stream-based communication is proposed in [62]. When the number of data exchanged between the processes is not known, i.e., data dependent applications, the data/credit parameters must be replaced by a dynamic stopping criteria. Thus, two approaches are presented: iteration-based and mode-based. In the first one, the producer process is responsible for signaling the end of data transfer process, while in the second one, an external controller is used to determine when the data transfers are finished. Besides the flexibility increase, the software support for this mechanism is not presented. Indeed, the external controller centralizes the configurations and is implemented in software, which might generate performance bottlenecks.

Addressing shared memory architectures, a hardware mechanism for tightly-coupled embedded systems is introduced in [63, 64]. The mechanism support data transfers

between processing cores, hardware accelerators and local memories inside a cluster by multiplexing data access requests to/from hardware accelerators in the shared memory. The software support is provided by functions for computer vision and image processing, which are accessible by the application through custom OpenMP extensions. However, the hardware mechanism provides support only at the local interconnection, without addressing the communication between processes executing in different clusters.

Finally, in [65], the adopted data communication is based on multilevel switching. Instead of moving the data, the cores can read from other cores directly by switching. Basically, the co-processors inside a core (group of a RISC CPU and several co-processors) can read data from other co-processor's memory space inside or outside their respective core. However, if the number of memory access increases at a remote core, the data is moved to the local core memory. Moreover, an independent thread is responsible for transferring data between the several memory levels (external, L3, L2 and L1). This thread can speculate and pre-fetch data, making available the needed data in each core as long as possible. The main drawback of the system is the programming model, which is a custom solution based on task level parallelism. In addition, the target architecture is a very specific solution for image processing applications, where the communication is handled by a switch, thus, limiting scalability.

Similarly to the previous section, the comparison between the described works and this thesis is performed in Table 1.3 (page 26).

1.3.3 Thesis Positioning

Table 1.3 presents the comparison between the hardware mechanisms described in Sections 1.3.1 and 1.3.2 regarding the objectives of this thesis. The goal is to provide higher flexibility (i.e., the possibility of handling a higher number of application scenarios) and develop a solution to address both communication aspects (synchronization/set-up and data transfer), while taking into account support for an inter-process communication standard, in this case, MCAPI.

It is possible to see in the comparison table that some works present solution for both communication aspects ([50, 61, 62]). However, the software support is either not addressed or is presented as a custom solution. On the other hand, MCAPI is targeted in [60]. Nonetheless, as already mentioned, the support for MCAPI channels is not presented.

The targeted flexibility is inspired by works such as [50], [52], [53], [59], [60] and [62]. To achieve this, the proposed mechanisms can be programmed through memory-mapped registers, which decreases programming complexity and increase flexibility. In addition, the mechanisms are co-designed with MCAPI in order to favor hardware-software interface. Furthermore, although the target software standard is MCAPI, the mechanisms can be employed by other solutions to decrease the software overhead due to their flexibility.

TABLE 1.3: Thesis positioning regarding communication and programmability aspects in relation to the state-of-the-art.

Reference	**Flexibility**	**SW API Support**	**Data Transfer HW Support**	**Synchronization HW Support**	**Target Architecture**
Calcado [49]	+	Custom	No	Yes	Tightly-Coupled Embedded Systems
Kachris [50]	++	Not Mentioned	Yes	Yes	Embedded Systems
Meyer [51]	+	MPI	No	Yes	SMP Clusters
Tabhet [52]	+++	RTM API	No	Yes	P2012 [27]
Reble [53]	+++	Custom MPI	No	Yes	SCC [14]
Kim [54]	+	Custom	No	Yes	Embedded Systems
Papadopoulos [55]	+	Not Mentioned	No	Yes	Embedded Systems
Han [56]	+	Custom	Yes	No	Embedded Systems
Buono [57]	+	Custom	Yes	No	Multi-threaded Multi-core
Gao [58]	++	Not Mentioned	Yes	No	Heterogeneous Clusters
Kumar [59]	++	Custom	Yes	No	Embedded Systems
Wallentowitz [60]	+++	MCAPI	Yes	No	Embedded Systems
Clermidy [61]	++	Custom	Yes	Yes	Embedded Systems
Helmstetter [62]	+++	Not Mentioned	Yes	Yes	Embedded Systems
Burgio [63, 64]	+	Custom/Open MP	Yes	No	Tightly-Coupled Embedded Systems
Ku [65]	+	Custom	Yes	No	Embedded Systems
This thesis	+++	MCAPI	Yes	Yes	Embedded Systems

More specifically, the communication set-up mechanism has two parameters that can be dynamically programmed: connection identifier and expected value. In this thesis, each connection identifier corresponds to an endpoint, while the expected value can be programmed according to the operation to be performed, e.g., wait the channel set-up to be completed. Also, the mechanism is able to store different values for each endpoint, making it possible for the processor to be notified at different events for each endpoint.

The mechanism responsible for the data transfer phase controls data exchange in a FIFO manner, as required by MCAPI channels. The flexibility concerns how the FIFOs

are managed and how the data transfers can be performed. Indeed, the mechanism does not provide dedicated FIFO, as presented by [56], but registers that store FIFO parameters, such as size and initial address. Therefore, the mechanism has no restriction regarding FIFO size, number of FIFOs (controlling more FIFOs requires more registers) or FIFO location, which can be placed in memory or implemented as dedicated resources for hardware accelerators. Since the application will not directly interact with the mechanism, it is up to the MCAPI implementation to define the amount of memory reserved for the FIFOs, which will impact the number and size of FIFOs. Finally, the data transfer can be performed in two ways: informing source/destination addresses or destination connection identifier, which increases flexibility.

Thus, the work presented in this thesis differs from the works previously described by providing hardware support for communication set-up and data transfer while targeting co-design with a standardized communication API.

1.4 Reference Architecture

A generic multi-core architecture is used as reference in order to implement and evaluate the proposed mechanisms. This architecture follows the most common characteristics presented in Section 1.1, such as a NoC interconnection and general purpose processors as main processing engines. The objective is to develop the mechanisms under conditions that can be met by most of the platforms, increasing their compatibility. Also, following the trend of distributed architectures, the reference architecture is divided in clusters, as depicted in Figure 1.7.

Each cluster comprises the CPUs (MIPS R3000 core) with their respective private memories and control registers (Ctrl Regs), input (In) and output (Out) NoC modules (CPU subsystem), a Shared Memory, a Network Interface (NI) and a DMA. Using this template, it is possible to have intra and inter cluster communication by using the shared memory and/or message passing. Additionally, hardware accelerators could replace some clusters in order to make the architecture heterogeneous.

Inside a cluster, the interface between the CPUs with the NI is performed by the modules *In* and *Out*. Both modules are shared between the CPUs and provide a queue to store the information to be sent to and from the CPUs, respectively. Thus, when the NI is busy and the CPU needs to send data through the NoC, it does not need to wait until the NI becomes available. The control registers are used to set/unset the CPUs to/from idle state and to store interrupt informations. The private memory stores the application code, while the Shared Memory is used to store shared data or implement

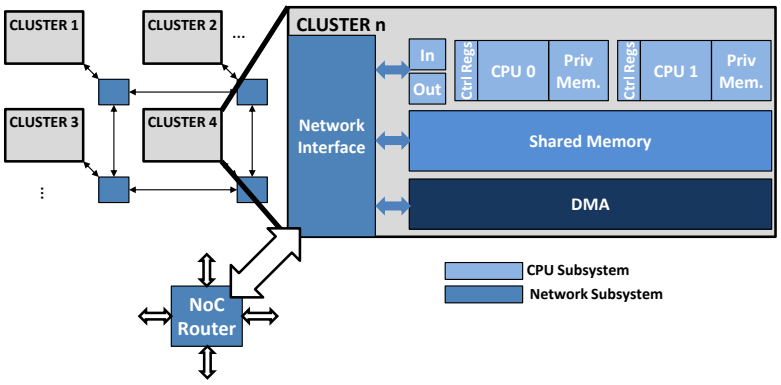


FIGURE 1.7: Reference architecture block diagram and hierarchy.

communication buffers. The DMA is able to perform data transfers through requests issued by the CPUs. These requests provide usual parameters: source buffer address, destination buffer address and transfer size. Finally, in order to avoid address translation when moving data between clusters, the address mapping is global. Therefore, the set of shared memories can be considered as a single distributed shared memory with non-uniform memory access time (NUMA).

The address map is detailed in Table 1.4. Thus, to read or write in a remote cluster, it is necessary to encode the respective identifier (value greater than zero) in the 9 MSBs (most significant bits) of the target address. On the other hand, when performing local read/write operations, the cluster identifier must remain zero.

TABLE 1.4: Cluster address map.

Cluster ID [31..23]	**Address (hex) [22..0]**		**Module**
	Start	*End*	
N	0x000000	0x01FFFC	Local Memory 0
	0x020000	0x03FFFC	Local Memory 1
	0x040000	0x13FFFC	Shared Memory
	0x140000	0x1FFFFC	Unused
	0x200000	0x20003C	Control Regs 0
	0x200040	0x20007C	Control Regs 1
	0x200080	0x2000FC	Unused
	0x200100	0x200114	Input 0
	0x200118	0x20012C	Input 1
	0x200130	0x2001FC	Unused
	0x200200	0x20023C	Output
	0x200240	0x2004FC	Unused
	0x200500	0x2005FC	DMA

The Network Interface is responsible for interfacing the cluster and NoC. The modules that can send data through the NI are CPUs and DMA. In turn, the data received

by the NI is forwarded to the respective module based on the destination address and address mapping. The NIs exchange data through the NoC in packets, which are composed by *flits* (32-bits words). Additionally, the NI can read and write data in the private and shared memories, similarly to a remote DMA. Table 1.5 presents the packet types that are managed by the NI.

TABLE 1.5: Packets used to exchange data and control through the NoC.

Packet Type	Description
WRITE REQ	Writes data in the respective destination address.
END OF WRITE	Signals the end of a move packet.
READ REQ	Request the data from a given address.
READ RESP	Replies with the data from the respective address.
CONTROL	Sends a 32-bit control word.
TEST AND SET	Performs an atomic test and set operation.

The data movement is performed by the `WRITE REQ` packet, which carries the initial destination address and the data. When sending multiple flits, the data is stored sequentially by incrementing the initial destination address. The `END OF WRITE` packet is used to inform that the data sequence is finished. Thus, large chunks of data can be split in several `WRITE REQ` packets to avoid network contention. The packets `READ REQ` and `READ RESP` are used to perform read operations in a remote address. Once a `READ REQ` is received, the NI access the respective requested address to read the data and packs it into a `READ RESP` packet. Then, when the response is received, the data is forwarded to the input (*In*) module. The `CONTROL` packet is used to send a single word of 32-bits directly to the input module of a given cluster. This packet can be used to send control information or to perform synchronization between tasks. Finally, the `TEST AND SET` packet is used to perform an atomic test and set operation in the memories. This packet is similar to a `READ REQ`, except that the operation can be performed only on memories. If the memory address had the logic value '0', it will be changed to logic value '1'. Otherwise, the value is not changed. In any case, a `READ RESP` packet containing the previous memory value is issued to the CPU that sent the `TEST AND SET` packet, i.e., logic value '0' means the memory address was set to '1'.

From the software point of view, a Hardware Abstraction Layer (HAL) is provided to program and access the hardware resources. As the peripherals are accessed as memory-mapped registers, the functions implement read and write in specific memory addresses. This allows the applications to abstract the address mapping and seamlessly access hardware resources. These functions include functionalities such as creating data transfer requests for DMA, sending the packets presented in Table 1.5, read data from the input module and access control registers.

In addition, the inter-process communication can be performed through FIFOs, which benefits a wide range of data-flow applications. The FIFOs are placed in the Shared Memory and controlled by software routines, called FIFO API. Thus, the application does not need to manage local or remote addresses in data transfers, decreasing programming complexity at application level. When a CPU communicates with another CPU in the same cluster, they use FIFOs placed in the local shared memory to exchange data. However, when CPUs in different clusters needs to communicate, a packet is sent to the FIFO placed in the destination cluster through the NoC, since reading data from a remote address is more costly than writing data in a remote address.

The FIFOs are controlled through a FIFO descriptor that contains the following fields: `base_address`, `size`, `first` (read pointer), `last` (write pointer) and `remote_addr`. The application must initialize the FIFO descriptor prior to writing and reading data. Figure 1.8 shows the steps performed in the initialization (1 and 2) and data exchange (3) in both communication sides. Initially, each process provides the address of their respective write/read pointers to be updated. Then, each process stores the remote pointer address in their `remote_addr` FIFO descriptor field. Later, when the data is exchanged, the pointers are written in the respective remote address, avoiding remote reads to check empty and full conditions.

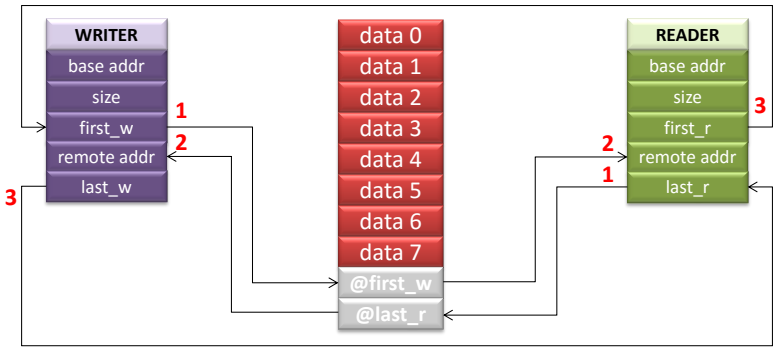


FIGURE 1.8: FIFO descriptor initialization and update.

Lastly, the application can read and write data by using three functions: `fifo_read`, `fifo_write` and `fifo_write_block`. The first two functions are used to read and write a single word of 32-bits in the FIFO and third function allows writing multiple words. Additionally, the `fifo_write_block` takes advantage of the DMA to transfer data, while the `fifo_write` uses the `WRITE REQ` packet to send data and update the remote write pointer. On the other hand, as the FIFOs are placed in the receiver cluster memory, the `fifo_read` is implemented as a simple read in the shared memory followed by a `WRITE REQ` packet to update the remote read pointer.

Chapter Summary

This chapter presented related works regarding multi-core architectures, software solutions for APIs and hardware mechanisms to increase synchronization and communication performance. However, unlike the previous works, this thesis focus on co-designing hardware mechanisms and a standardized software API for a generic multi-core architecture. The objectives of this approach are to increase programmability and communication performance. Also, the mechanisms are flexible and aim to decrease programming complexity.

Chapter 2

MCAPI Mapping and Overhead Characterization

This chapter presents the MCAPI standard and its implementation for the reference architecture, which is the first contribution of this Thesis. Section 2.1 describes the MCAPI specification and how the standard can be used. Next, Section 2.2 explains the design choices performed to implement the MCAPI specification (version 2.015) for the reference architecture and details the implemented functions. Finally, an analysis is carried out in Section 2.3 in order to evaluate the main drawbacks and bottlenecks regarding the implementation for the reference architecture, which are caused mainly due to lack of hardware support.

2.1 MCAPI Standard and Specification

MCAPI [6] is a specification for inter-task communication and synchronization between processing cores in embedded systems. The main goal of MCAPI is to provide source code portability, scalable communication performance and low memory footprint. Additionally, the specification does not restrict the system topology, which can be either homogeneous or heterogeneous architectures located on a single chip or on multiple chips in a circuit board.

The MCAPI specification defines two levels of hierarchy: **domains** and **nodes**. A **domain** is composed of one or more **nodes** and it is used for routing purposes. Its scope is defined by each implementation and might be a set of processors in a multi-core system, or even an entire circuit board with several chips. A **node** is defined by the specification as an "independent thread of control" [6], i.e. an entity that can exclusively

execute a sequential flow of instructions, such as a process, thread, processor, hardware accelerator, etc. The definition's objective is to have a single **node** definition within the same implementation. **Domains** and **nodes** have a global unique identifier, which means that **domain** numbers must not be repeated, as well as **node** numbers that are grouped in the same **domain**.

The communication is performed between **nodes** through a pair of socket-like termination points called **endpoints**. The **endpoints** can be created and managed by the application through specific functions. The maximum number of **endpoints** that a given **node** can have must be defined prior to the compilation. Each **endpoint** is created by passing a port identifier as argument to the respective function, and can be managed by several functions by passing as argument a tuple composed of **domain**, **node** and port identifiers. Additionally, each **endpoint** has its own set of attributes related to Quality of Service (QoS), timeouts, buffers, etc. As the **endpoints** are unidirectional, the communication is often referred to occur between a *sender* and a *receiver*. Three communication modes are supported (Figure 2.1):

Messages – Connection-less datagrams.

Packet Channels – Connection-oriented, unidirectional, FIFO packet streams.

Scalar Channels – Connection-oriented, single-word, unidirectional, FIFO packet streams.

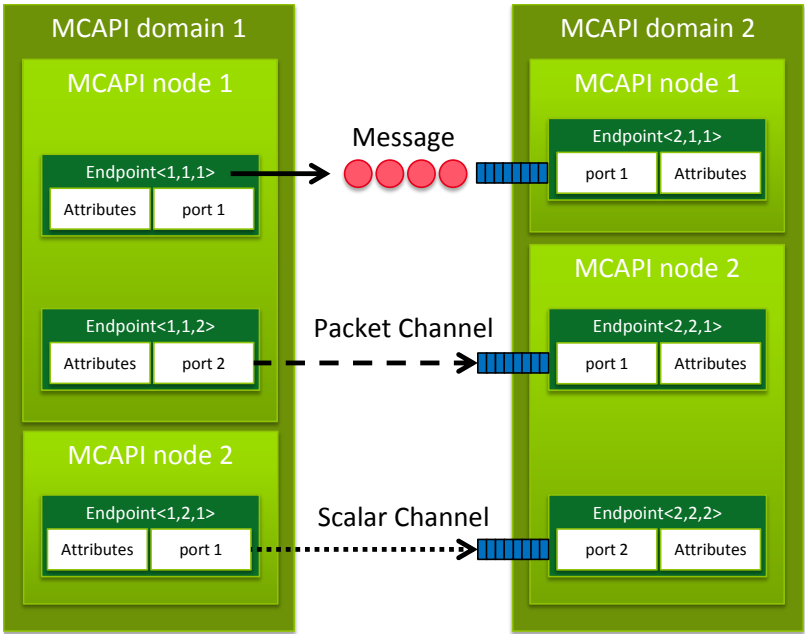


FIGURE 2.1: MCAPI communication modes.

The upper part of Figure 2.1 depicts messages transmission between the **endpoints** <1,1,1> and <2,1,1>. Messages are similar to UDP datagrams in networking. The main difference between messages and packet or scalar channels is the flexibility, allowing data transmission between *sender* and *receiver* without having to establish a connection. Other differences are the possibility to send messages with different priorities and the sender and receiver buffers, which must be provided by the application.

Packet and Scalar channels are depicted in Figure 2.1 in the transmissions between the pair of **endpoints** <1,1,2> and <2,2,1> and the pair of **endpoints** <1,2,1> and <2,2,2>, respectively. The main difference between packets and scalar channels is the size of data transfers. Packet channels are able to transfer data chunks of variable sizes, while Scalar channels support transfers of 8, 16, 32 or 64-bits only. Both Channel types are able to provide lightweight socket-like stream communication by establishing an unidirectionally connection between *sender* and *receiver* **endpoints** previously to data transfer. The data is delivered in a FIFO manner and, contrarily to Messages, the buffer in the *receiver* side is provided by the MCAPI implementation (the *sender* buffer must be provided by the application). However, this buffer must be returned to the MCAPI implementation once the *receiver* has finished to process the received data.

The connection set-up is performed in two steps for both Packet and Scalar channels: *connection* and *opening*. These two steps must be performed by both sides of the communication channel. Later, when the communication is finished, both sides must perform a *closing* step to release the communication buffer and other possible allocated resources. Collective operations such as multi or broadcast are not currently supported by the MCAPI specification.

The MCAPI functions defined by the specification for establishing a connection (packet and scalar channels) are only non-blocking. On the other hand, send and receive functions for Messages and Packet Channel have blocking and non-blocking variants. The non-blocking functions have a "_i" appended to the function name to indicate that the function will return immediately. The non-blocking functions return a `request` structure that can be used to query its status. Three operations can be performed using a `request` structure:

`test` - Verifies if the respective function of a given `request` has completed.

`wait` - Waits until the respective function of a given `request` has completed.

`cancel` - Cancels the execution of the respective function of a given `request`.

The `test` and `cancel` operations are non-blocking, returning to the application upon finishing the `request` processing, while the `wait` operation will block until the requested function completes or a time-out occurs.

The most common MCAPI functions are presented in Table 2.1. Since scalar channels are intended to provide very low-overhead for moving a stream of values and non-blocking functions add overhead, send and receive operations are available only as blocking functions. As previously mentioned, the non-blocking functions are identified by the suffix "_i" in their names, e.g., `mcapi_pktchan_send_close_i` and `mcapi_pktchan_connect_i`.

2.2 MCAPI Implementation

The first contribution of this Thesis is the implementation of the MCAPI specification for the reference architecture presented in Section 1.4. This Thesis focuses on the implementation of packet channel communication mode. This can be explained by the fact that packet channels are more flexible than scalar channels and provide better performance than messages, covering a wider range of data-flow applications. Moreover, due to characteristics such as resource reservation and FIFO-like data exchange, there are more opportunities to exploit hardware mechanisms than the other communication modes. Additionally, applications can communicate using only the set of functions provided by packet channels, since the data types used in scalar channels can be sent through packet channels and messages are mostly used for synchronization and initialization. The MCAPI **domains** and **nodes** are represented by clusters and CPUs in the reference architecture, as showed in Figure 2.2.

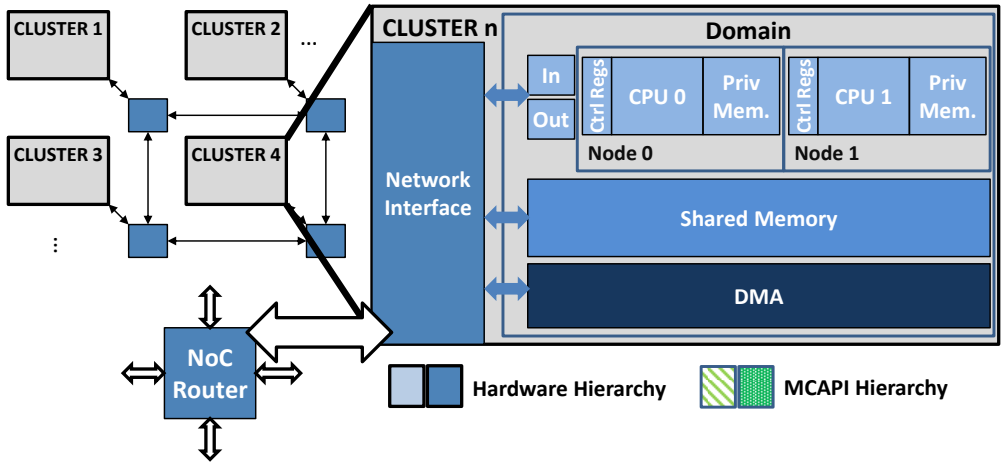


FIGURE 2.2: MCAPI domain and node mapping in the reference architecture.

TABLE 2.1: Non-exhaustive list of MCAPI functions.

Group	Function	Description
General	`mcapi_initialize`	Initializes an MCAPI node.
	`mcapi_finalize`	Finalizes the MCAPI environment on the local MCAPI node.
Endpoints	`mcapi_endpoint_create`	Creates an endpoint in the local node with the specified port ID.
	`mcapi_endpoint_get`	Retrieves and endpoint in a remote node regarding the informed domain, node and port identifiers.
	`mcapi_endpoint_delete`	Delete the specified endpoint in the local node.
Messages	`mcapi_msg_recv`	Receives a message from a receive endpoint.
	`mcapi_msg_send`	Sends a message from a send endpoint to a receive endpoint.
Packet Channel	`mcapi_pktchan_connect_i`	Connects a pair of endpoints into an unidirectional FIFO packet channel.
	`mcapi_pktchan_open_recv_i`	Opens the receive end of a packet channel.
	`mcapi_pktchan_open_send_i`	Opens the send end of a packet channel.
	`mcapi_pktchan_recv`	Receives a packet on the opened packet channel.
	`mcapi_pktchan_send`	Sends a packet on the opened packet channel.
	`mcapi_pktchan_available`	Checks if packets are available on a receive endpoint.
	`mcapi_pktchan_release`	Returns the system buffer for the MCAPI implementation after finishing a packet receive call.
	`mcapi_pktchan_recv_close_i`	Closes the receive side of a packet channel.
	`mcapi_pktchan_send_close_i`	Closes the send side of a packet channel.
Scalar Channel	`mcapi_sclchan_connect_i`	Connects a pair of endpoints into an unidirectional FIFO scalar channel.
	`mcapi_sclchan_recv_open_i`	Opens the receive end of a scalar channel.
	`mcapi_sclchan_send_open_i`	Opens the send end of a scalar channel.
	`mcapi_sclchan_recv_uintX`[a]	Receives a X-bit scalar on an opened scalar channel.
	`mcapi_sclchan_send_uintX`[b]	Sends a X-bit scalar on an opened scalar channel.
	`mcapi_sclchan_available`	Checks if scalars are available on a receive endpoint.
	`mcapi_sclchan_recv_close_i`	Closes the receive side of a scalar channel.
	`mcapi_sclchan_send_close_i`	Closes the send side of a scalar channel.
Non-Blocking Operations	`mcapi_test`	Checks if a non-blocking operation has completed.
	`mcapi_wait`	Wait until a non-blocking operation has completed.
	`mcapi_cancel`	Cancels an outstanding non-blocking operation.

[a] X stands for values 64, 32, 16 and 8.
[b] Refer to footnote *a*.

This mapping was defined based on the fact that each CPU can execute only one task (thread) at a time, which configures an independent thread of control. Moreover, as the **domains** are used for routing purposes, CPUs in the same cluster can be grouped in a single **domain**, since the data does not have to be routed by the NoC. When communicating with a CPU located in a remote cluster, the MCAPI implementation must compute the path to the target cluster, characterizing a communication between two different **domains**. Each **node** has its own set of **endpoints**, which are represented by a 32-bit integer that encodes cluster, **node** and port identifiers. The codification has the 16 most significant bits (MSBs) representing the **domain** identifier, the following 8 MSBs representing the **node** identifier and the 8 less significant bits (LSBs) representing the port identifier.

2.2.1 Data Structures

In order to implement MCAPI for the reference architecture, some information have to be accessible by all MCAPI domains. This is achieved by statically allocating areas of the Shared Memory in each cluster to store three data structure types: MCAPI attributes, FIFOs and Requests. The main advantage of storing this information in the Shared Memory is allowing remote **nodes** to easily read the data by performing a simple read in the remote Shared Memory.

2.2.1.1 MCAPI Attributes Structure

The organization of **domain**, **node** and **endpoint** attributes is presented in Figure 2.3. The **domain** data is placed in the top of the structure and contains all its **node** structures. The address storing the **domain** identifier is called **base address** and is defined prior to compilation. The **node** structures are organized in an ascending order (lowest identifier in the lowest address and highest identifier at the highest address). Each **node** structure has its own attributes and the attributes of its own **endpoints**. The **endpoint** attributes comprise maximum payload size, status, buffer type, FIFO descriptor (Section 1.4, Figure 1.8), among others. Since the maximum number of **endpoint** is fixed, **domain**, **node** and **endpoints** attributes can be accessed by simply adding an offset to the **base address** based on **node** and port identifiers. As an example, considering that Node 0 needs to read the attribute 5 of **endpoint** 8 at Node 1 in the following configuration:

- **base address** at 0x0004000;
- Size of **domain** attributes: 16 bytes (0x10);

– Size of **node** attributes: 16 bytes (0x10);

– Size of **endpoint** structure: 32 bytes (0x20);

– Size of each **endpoint** attribute: 4 bytes (0x04);

– Maximum number of **endpoints**: 16;

The address that should be read is the 0x0004344, since it is needed to sum up:

$$\begin{aligned} read\_addr = base\_address + domain\_attributes \\ + (node\_attributes + (endpoint\_structure * maximum\_endpoints)) \\ + node\_attributes + (endpoint\_structure * endpoint\_port\_id) \\ + (attribute\_size * attribute\_number) \end{aligned} \tag{2.1}$$

When an attribute located in a remote **domain** has to be read, the address calculation is performed in the same way, with the **domain** identifier being used to properly address the remote cluster in the MSBs of the address, as mentioned in Section 1.4. The attributes are accessed most of the time when a packet channel connection is being set-up, since several endpoint attributes must match in order to successfully establish the connection.

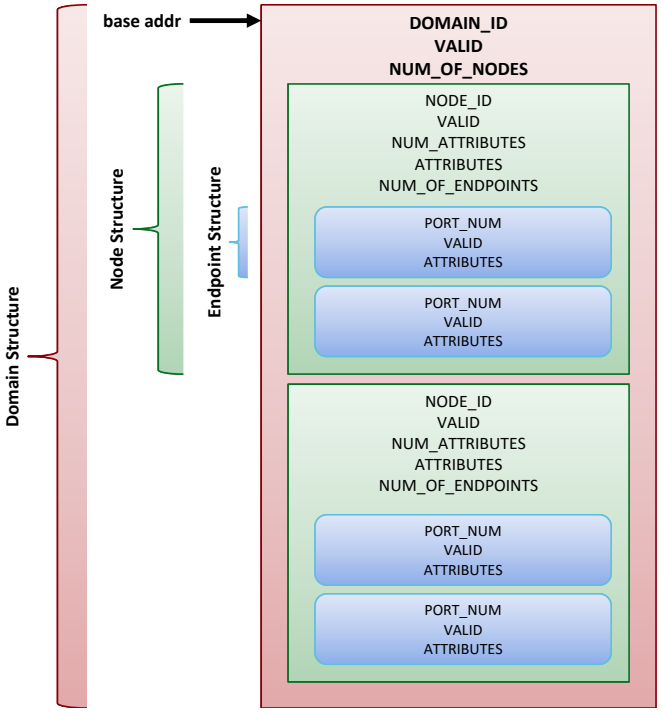


FIGURE 2.3: MCAPI attributes organization in the Shared Memory.

2.2.1.2 FIFO Structure

The packet channels require the data to be transmitted in a FIFO manner. To accomplish this with limited overhead, an area of the Shared Memory is dedicated to implement FIFO structures, based on the available FIFO API presented in Section 1.4. This structure is depicted in Figure 2.4 and is responsible for storing the data transmitted between *sender* and *receiver*, the addresses of read and write pointers (first and last), the endpoint identifiers of *sender* (s_endpt) and *receiver* (r_endpt) processes and a "lock" variable used to inform if the respective structure is being used. Therefore, the MCAPI implementation can take advantage of FIFO API functions to implement packet channel transmission. The FIFOs are shared among all **nodes** of a Cluster. Since the FIFO control is implemented in software, the number of FIFOs and their respective sizes are flexible and can be tuned according to the requirements of each application.

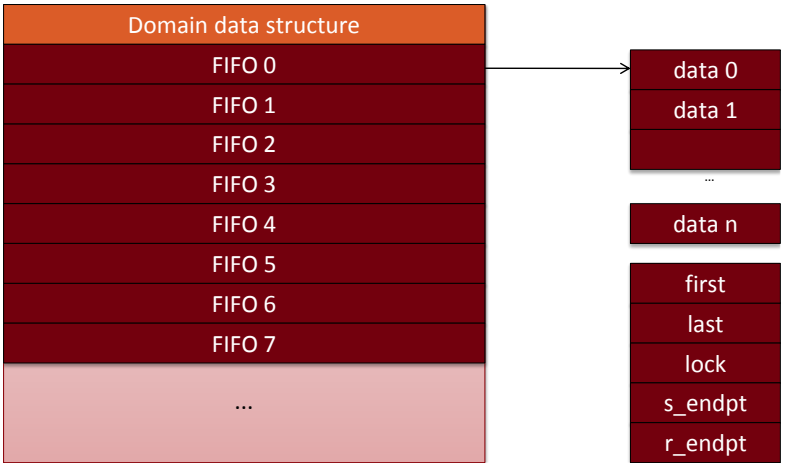


FIGURE 2.4: Organization of the FIFO structures in the Shared Memory.

The FIFO is allocated in the *connection* step and is always placed at the receiver side. This design choice is due to the fact that remote writes (posted write) present lower latency over remote reads, since the waiting time of a response is removed. In order to implement an atomic operation in the FIFO lock variable, the test-and-set packet (Table 1.5) is used to perform the FIFO allocation. Then, the FIFO structure is initialized with write and read pointers initial values and endpoints identifiers. Later, during the data transfer phase, the data is exchanged in the respective FIFO addresses in the Shared Memory and the pointers are updated in both *sender* and *receiver* sides accordingly. Finally, in the *closing* step, the FIFO structure is released by updating the lock variable and reseting read/write pointers and endpoint identifiers.

2.2.1.3 Request Structure

The last structure created to support the MCAPI implementation is related to the `request` structure used to query the status of non-blocking operations. The implementation provides a programmable number of `requests` per **endpoint**. Table 2.2 presents the fields that compose a `request` structure and its respective descriptions. All the `requests` structures are placed in the Shared Memory. It is important to highlight that the organization of these structures in the Shared Memory can be modified to present higher efficiency in terms of memory usage.

TABLE 2.2: Request structure fields.

Request Field	Description
`id`	Request identifier.
`function`	The function related to the respective request.
`endpt1`	Sender endpoint identifier.
`endpt2`	Receiver endpoint identifier.
`fifo_id`	Identifier of the FIFO structure.
`size`	Amount of data exchanged in bytes.
`status`	Request status.

The structure must be filled during the execution of a non-blocking function by the MCAPI implementation. Some fields are not used by all functions, such as `size` and `fifo_id`. The `id` field is used to calculate the offset needed to access the respective structure in the Shared Memory. The `function` field is responsible for encoding the function that is related to the respective `request` and is used to decide which actions should be performed by non-blocking operations. The endpoint identifiers are used to access the **endpoints** related to the function and, eventually, access their attributes and update them, e.g when a `mcapi_pktchan_send_close_i` has finished and the endpoint status has to be changed from MCAPI_CLOSE_PENDING to MCAPI_AVAILABLE. The `fifo_id` field is filled only by packet channel non-blocking functions and is used to determine the offset needed to access the respective FIFO in the Shared Memory. The FIFO structure is accessed during the *connection* and *closing* steps in order to check or modify the lock variable. The `size` field is used to return to the application the amount of data exchanged during non-blocking packet channel send or receive calls. When other non-blocking functions are executed this value will be always zero. Finally, the `status` field informs the current status of the respective `request` and is also used by non-blocking operations to determine the actions that must be performed.

2.2.2 Connection Set-up non-blocking Functions

The implementation of packet channel non-blocking functions focus on allocating resources in the *connection* step and updating endpoint attributes in the *opening* and *closing* steps. The common constraint is that all the steps must fill the `request` structure in order to successfully finish its execution during `wait` or `test` non-blocking operations. Furthermore, since all the steps must be performed by both sender and receiver **endpoints**, they can be seen as a handshake protocol for starting and finishing the communication. An example of the function call order for both sender and receiver processes is presented in Figure 2.5.

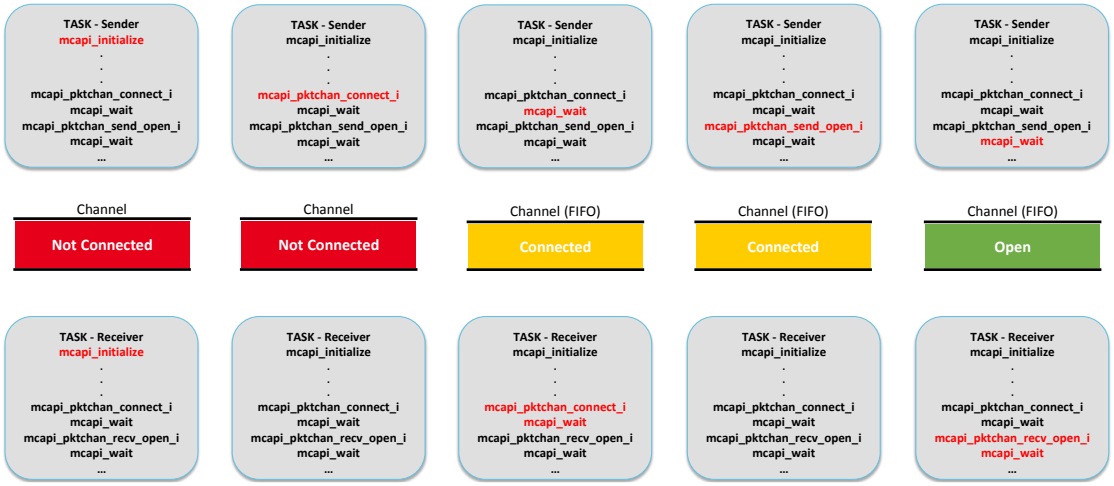


FIGURE 2.5: Function call order for a packet channel set-up.

The *connection* step is performed by calling the `mcapi_pktchan_connect_i` function. At the sender side, the MCAPI implementation seeks for a FIFO in the receiver **domain** using the test-and-set packet, which returns a positive value if an available FIFO was found or otherwise, the zero value. If a FIFO was successfully allocated, its structure is initialized as explained in Section 1.4 and a `request` structure is filled accordingly. Contrarily, if there is no available FIFO, an error is returned to the application. At the receiver side, the MCAPI implementation seeks for a FIFO in its local **domain** that has been initialized with the respective receiver endpoint identifier and fills a `request` structure accordingly. If there is not any FIFO corresponding to the receiver endpoint identifier, the `request` is filled with an invalid FIFO identifier.

The *opening* step is performed by calling the `mcapi_pktchan_send_open_i` and `mcapi_pktchan_recv_open_i` functions at sender and receiver side, respectively. These functions change the endpoint status to `OPEN_PENDING`, signaling that the *opening* step was initiated in that respective communication side. The handshake protocol is performed by non-blocking operations, and the implementation assumes that the receiver

side must start the handshake by changing its endpoint status to `OPEN` when it identifies that the sender status is `OPEN_PENDING`. Next, the receiver side waits until the sender status changes to `OPEN` to finish the protocol. The sender side only needs to wait the status of the receiver side to be `OPEN` in order to also modify its own status to `OPEN` and finish the protocol.

The *closing* step is similar to the *opening* step in terms of handshake protocol implementation. In addition, this step is also responsible for deallocating the FIFO employed in the communication channel. This step is performed by calling the `mcapi_pktchan_send_close_i` and `mcapi_pktchan_recv_close_i` functions at sender and receiver side, respectively. The endpoint status are changed to `CLOSE_PENDING` and the non-blocking operations perform the handshake protocol. When the handshake protocol is finished, the receiver side deallocates the respective FIFO structure by reseting the endpoint identifiers and write/read pointers and by releasing the lock variable.

Finally, although Figure 2.5 shows the sender process performing the set-up steps before the receiver, the implementation does not impose this order as a constraint. Indeed, the only order that must be respected is the step order, i.e., *connection* before *opening*, and *opening* before *closing*. However, within each step, the implementation supports both sender and receiver initializing the respective step.

2.2.3 MCAPI non-blocking Operations

Non-blocking operations are used to query the status of a `request` (`mcapi_test`) or to wait until the non-blocking function related to the respective `request` has finished (`mcapi_wait`), as mentioned in Section 2.1. These operations perform different actions for each connection set-up step and may return different request status. The `mcapi_wait` may return request success, invalid request or timeout status. In addition, `mcapi_test` may return also a request pending status, i.e the non-blocking function is not finished yet. The actions that must be executed depending on the returned status are defined by the application programmer. In either way, a communication set-up step is considered to be finished only when the returned status is success, i.e., it is mandatory to perform `mpcai_wait` or `mcapi_test` operations before executing the next communication set-up step, as depicted in Figure 2.5.

In the *connection* step, the non-blocking operations ensure that both communication sides had successfully allocated the FIFO and both endpoints changed their status. Firstly, the non-blocking operations have to identify the local **domain** communication side. In the sender side, the `mcapi_wait` operation checks the receiver endpoint status and, once it is connected, changes the sender endpoint status to connected. Similarly,

the `mcapi_test` operation checks the receiver **endpoint**, but returns a request pending status if it is not connected. In the receiver side, the first action is to check if the sender has allocated any FIFO related to the receiver **endpoint**. If there is not any FIFO related to the receive **endpoint**, the `mcapi_wait` operation will block until the FIFO is allocated, and the `mcapi_test` operation returns a request pending status. On the other hand, if a related FIFO has been identified, the receiver endpoint attributes are updated and the endpoint status is changed to connected. Then, the `mcapi_wait` operation blocks until the sender endpoint status is also changed to connected and returns a request success status, while the `mcapi_test` operation returns a request pending status. Finally, the `mcapi_test` operation returns a request success status when the sender endpoint status is connected.

The *opening* and *closing* steps are simpler than the *connection* step, since they only implement a handshake protocol, as mentioned in paragraphs 3 and 4 of Section 2.2.2. Thus, the `mcapi_wait` operation has to check the remote endpoint status and blocks if the status is not updated or, if the remote endpoint status is updated, changes the local endpoint status and blocks until the next handshake. Similarly, the `mcapi_test` operation has to check the remote endpoint status, change the local endpoint status if needed, and return request pending status while the handshake protocol is not finished.

2.2.4 Data Transfer

After performing the channel set-up steps, the data transmission may start. The packet channel function used to send data is the `mcapi_pktchan_send` and the function used to receive data is the `mcapi_pktchan_recv`. The send and receive calls receive a parameter called *handle*, which is a type defined by the MCAPI specification, but is implementation-specific. In this work, this type refers to the local endpoint structure in the Shared Memory (Section 2.2.1.1).

Despite the *handle*, the send function also receive a pointer to the source buffer and the amount of that must be transmitted in bytes as parameters. In this Thesis, the packet size is the first data sent in a packet channel data transfer. This decreases the receive function complexity, since it knows the expected size beforehand. Additionally, as the FIFO is already allocated, the MCAPI implementation of this function has only to retrieve the FIFO ID to calculate the offset to where the data should be sent (stored as an endpoint attribute in the Shared Memory) and take benefit from the functions provided by the FIFO API to send the data (write in the FIFO and check full status).

Similarly, the receive function also take benefit from the functions provided by the FIFO API to receive the data (read from the FIFO and check empty status). However,

the amount of received data and the buffer containing the received data are supplied to the application by the MCAPI implementation. In order to implement this specification constraint, the first received data is stored as the packet size and is used by the implementation to control the amount of data to be read from the FIFO. Furthermore, an area of the Shared Memory is also reserved for the so-called MCAPI System Buffer. In this Thesis, one System Buffer of 4 kB is available at each cluster. The main differences between the packet channel FIFOs and the System Buffer are that the later can be accessed and used by the application in any way it chooses to, and their storage capacity. While FIFOs are kept small to avoid hardware overhead (128 Bytes in the current implementation), the System Buffer size must be able to store the maximum size of a MCAPI packet, which is also implementation-specific. Thus, when a `mcapi_pktchan_recv` is performed, only the actual received data is copied from the FIFO to the System Buffer, i.e., the first received data is returned to the application as the amount of received data. Later, when the packet is finished and the application has already processed the data stored in the System Buffer, the application must return the System Buffer to the MCAPI implementation through the `mcapi_pktchan_release` function.

2.3 Performance Limitations of Software Implementation

The main advantage of fully implementing MCAPI in software is the flexibility, since there is no need to concern about the hardware architecture. Moreover, as mentioned in Chapter 1, the programming complexity might increase significantly when specific hardware solutions have to be used by the software layer, since the programmer must be aware of all hardware details. However, the software implementation might present significant overheads. In order to avoid these overheads, the MCAPI implementation was analyzed in two points: *Communication Set-up* and *Data Transfer*.

2.3.1 Communication Set-up Overheads

During this communication phase no data is transferred and still several packets are exchanged in order to verify endpoint status and attributes. A scenario containing one sender and one receiver is evaluated to characterize the traffic generated by the MCAPI implementation, with the results presented in Figure 2.6.

The result shows that, despite the `mcapi_pktchan_connect_i` function on both communication sides, the highest overhead is generated by the `mcapi_wait` operations, which is around 10 flits for each call. This overhead is explained by the polling performed in the remote address storing the remote endpoint status. Although this amount of flits

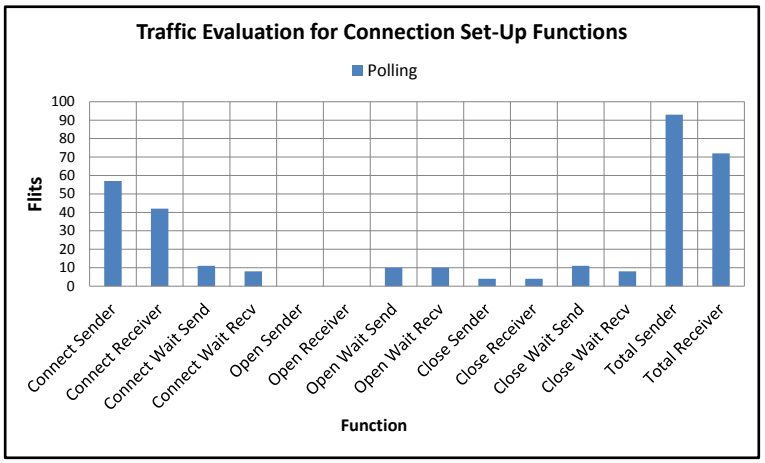


FIGURE 2.6: Evaluation of the number of flits sent by each function in the connection set-up.

might be not relevant when compared to the data transfer process, a condition herein called "*synchronization gap*", which is depicted in Figure 2.7, may significantly increase this overhead.

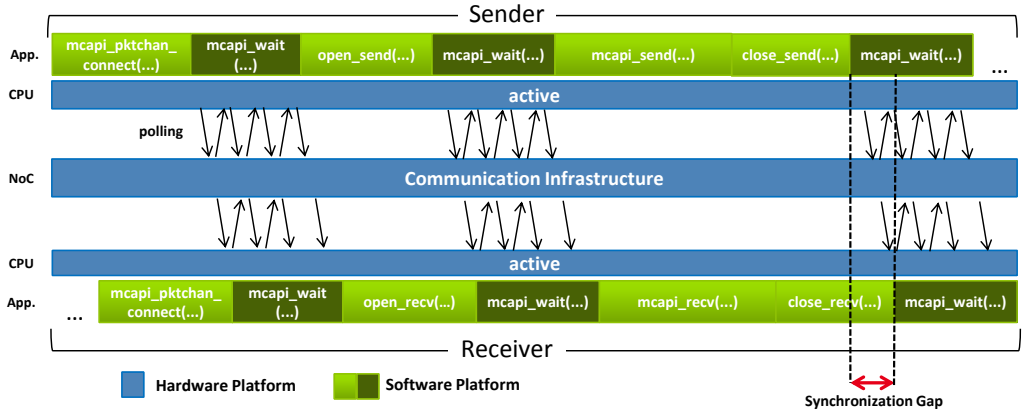


FIGURE 2.7: Synchronization gap between the connection set-up steps.

This gap is a desynchronization between sender and receiver and may occur due to several factors, such as different initialization times in the processors, different processing loads, etc. Moreover, it is a common scenario and it is mostly likely to occur when executing several applications. Therefore, in order to measure the impact of this condition, a scenario with one sender and one receiver performing the connection set-up 10 times was evaluated for several values of "desynchronization". This parameter was calculated by measuring the average execution time of the non-blocking functions used during the connection set-up phase. Then, the receiver side execution was delayed by different fractions of this value, which it is called "desynchronization rate". As an example, if the *opening* function is executed in the receiver side only after the equivalent

function was executed in the sender side, the "desynchronization rate" will be 100%, while functions that start to execute at the same time will have a "desynchronization rate" of 0%. The results are presented in Figure 2.8.

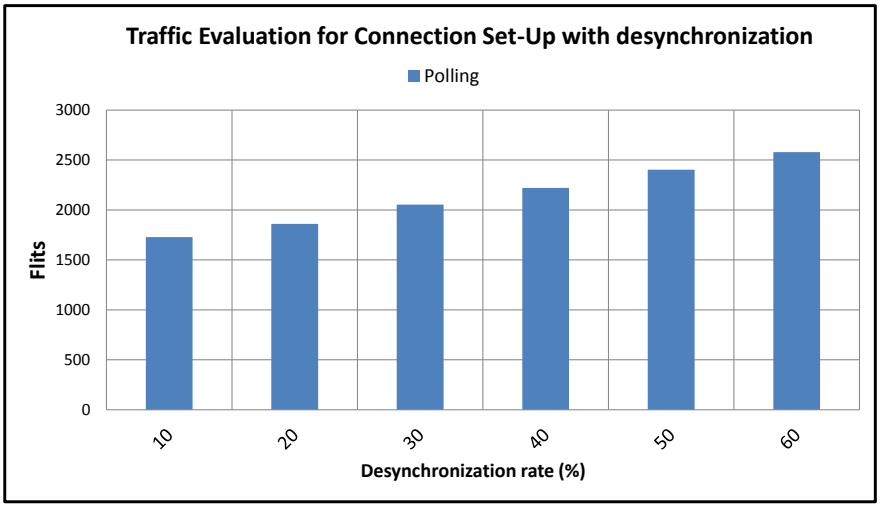


FIGURE 2.8: Generated traffic during channel set-up with desynchronization between sender and receiver.

The above chart shows that the traffic increase is linear as the desynchronization increases. Furthermore, the generated traffic with desynchronization of 60% is around 50% higher than the traffic generated with desynchronization of 10%. In addition to the increased traffic overhead, Figure 2.7 also shows that the CPU is always active, leading to processing overhead as well. Therefore, these numbers reflect the importance of providing an efficient solution that can be able to decrease these overheads. This issue is further discussed in Chapter 3, where a hardware mechanism that aims to decrease this overhead is proposed.

2.3.2 Data Transfer and FIFO Control Overheads

The data transfer phase is executed almost entirely by one function in each communication side (packet channel send and receive). Thus, the investigation of possible overheads can be narrowed to these functions. As already mentioned, both send and receive functions take advantage of the FIFO software implementation. However, the implementation of a FIFO mechanism in software creates processing and traffic overheads. Although the data transfer is performed by the DMA, the FIFO control is handled by the CPU and impact the performance during data transfers, as exemplified in Figure 2.9.

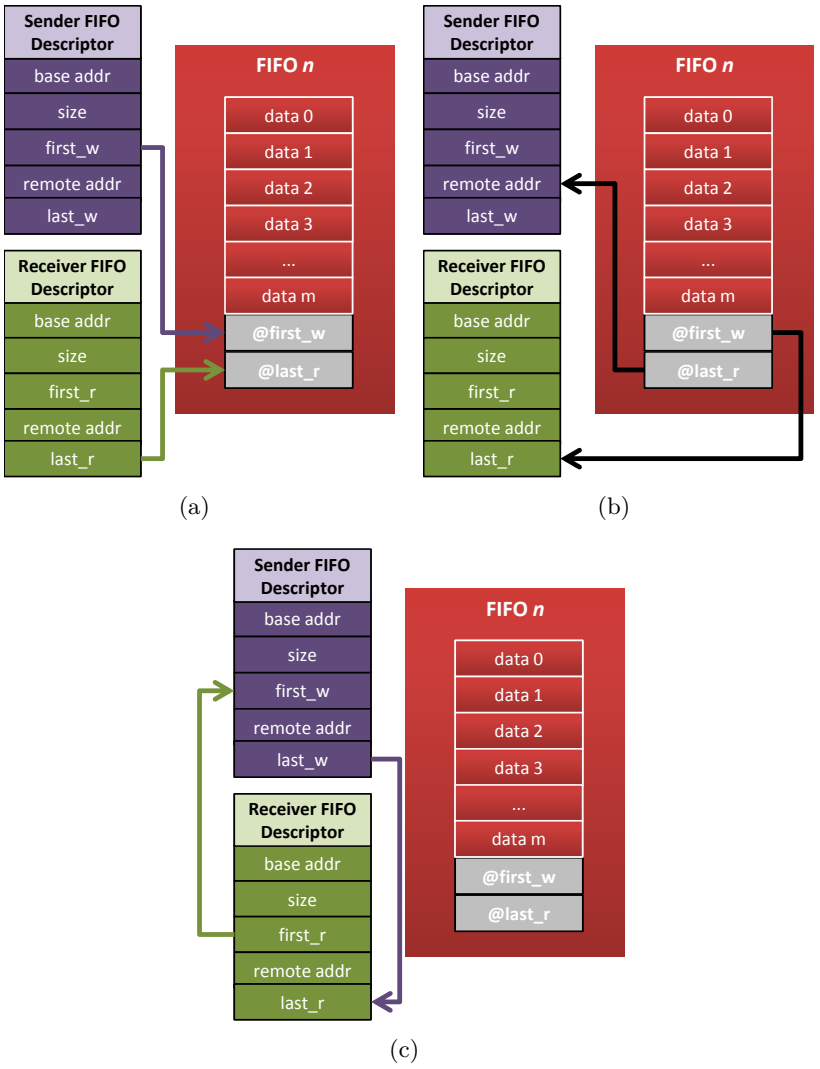


FIGURE 2.9: Initialization and pointer exchanging of FIFO structure.

In this example, the structure is initialized in the steps 2.9(a) and 2.9(b). In 2.9(a) the address of the sender process read pointer and the address of the receiver process write pointer are stored in the FIFO structure. Then, in 2.9(b), these addresses are stored in the field `remote_addr`, allowing each **endpoint** to update the remote pointer when the local pointer is updated (2.9(c)). Therefore, for each write or read operation, the CPU has to perform pointer updates and FIFO status checking. Moreover, when sender and receiver processes are in different clusters, the pointers must be exchanged through the NoC, resulting in traffic overhead and higher communication latencies.

Thus, in order to completely decouple computation and communication and increase system performance, a hardware mechanism to manage buffers/FIFOs with flexible configuration and low control overhead is proposed in this Thesis and is detailed in Chapter 4.

Chapter 3

Communication Set-up Support

This chapter presents the Event Synchronizer hardware module, which is the second contribution of this Thesis. The mechanism focus on decreasing processing and traffic overheads imposed by the MCAPI implementation. This is achieved by replacing polling phases with programmable event signaling. Moreover, as the mechanism was developed in co-design with MCAPI, it also targets to be flexible and easily programmable. The polling phases present in the MCAPI implementation are detailed in 3.1, while the Event Synchronizer is described in Section 3.2. Finally, the modifications performed in the MCAPI implementation to take advantage of the Event Synchronizer are presented in Section 3.3.

3.1 Communication Set-up Polling Phases

The software implementation of the communication set-up phase relies mainly on reading specific attributes of an endpoint until it matches an expected value (polling phase). Indeed, a given process can check many times a specific attribute of its target endpoint; if the target endpoint is in a remote cluster, several messages are sent over the NoC, requesting the value stored in that respective address. If in the same cluster, several read operations are performed in the Shared Memory. Either way, both cases present an overhead since the result of read operations remains the same until the endpoint status changes.

The polling phases are performed in both sides of the communication channel. Figure 3.1 depicts the polling phases performed during the communication set-up in the sender side. It is possible to see that four polling phases are performed from `mcapi_initialize` to `mcapi_finalize`. Considering only the `mcapi_wait` operation,

there are three polling phases, being one for each connection set-up step (*connect, open, close*).

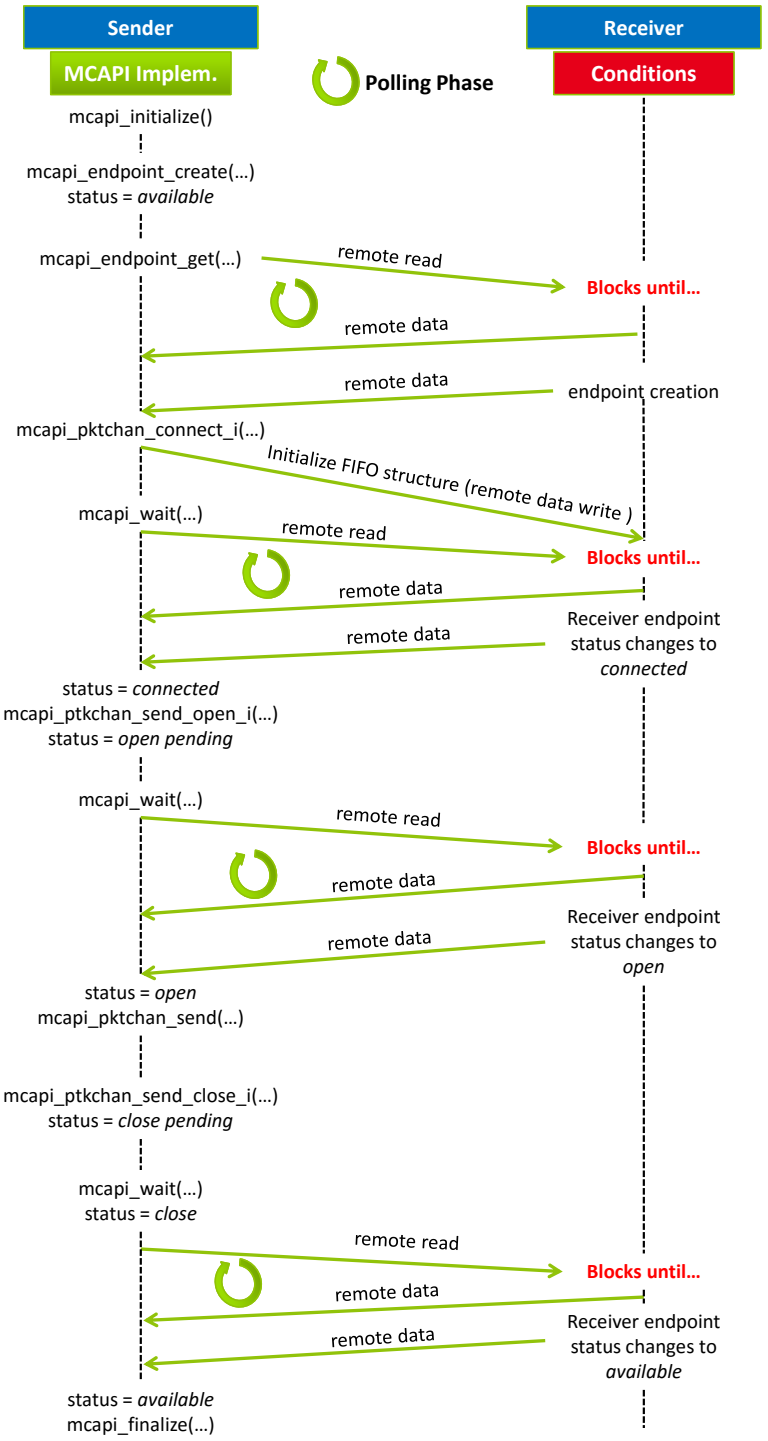


FIGURE 3.1: Connection set-up polling phases in the sender side.

At the receiver side, considering only the `mcapi_wait` operations, there are four remote and one local polling phases, as represented in Figure 3.2.

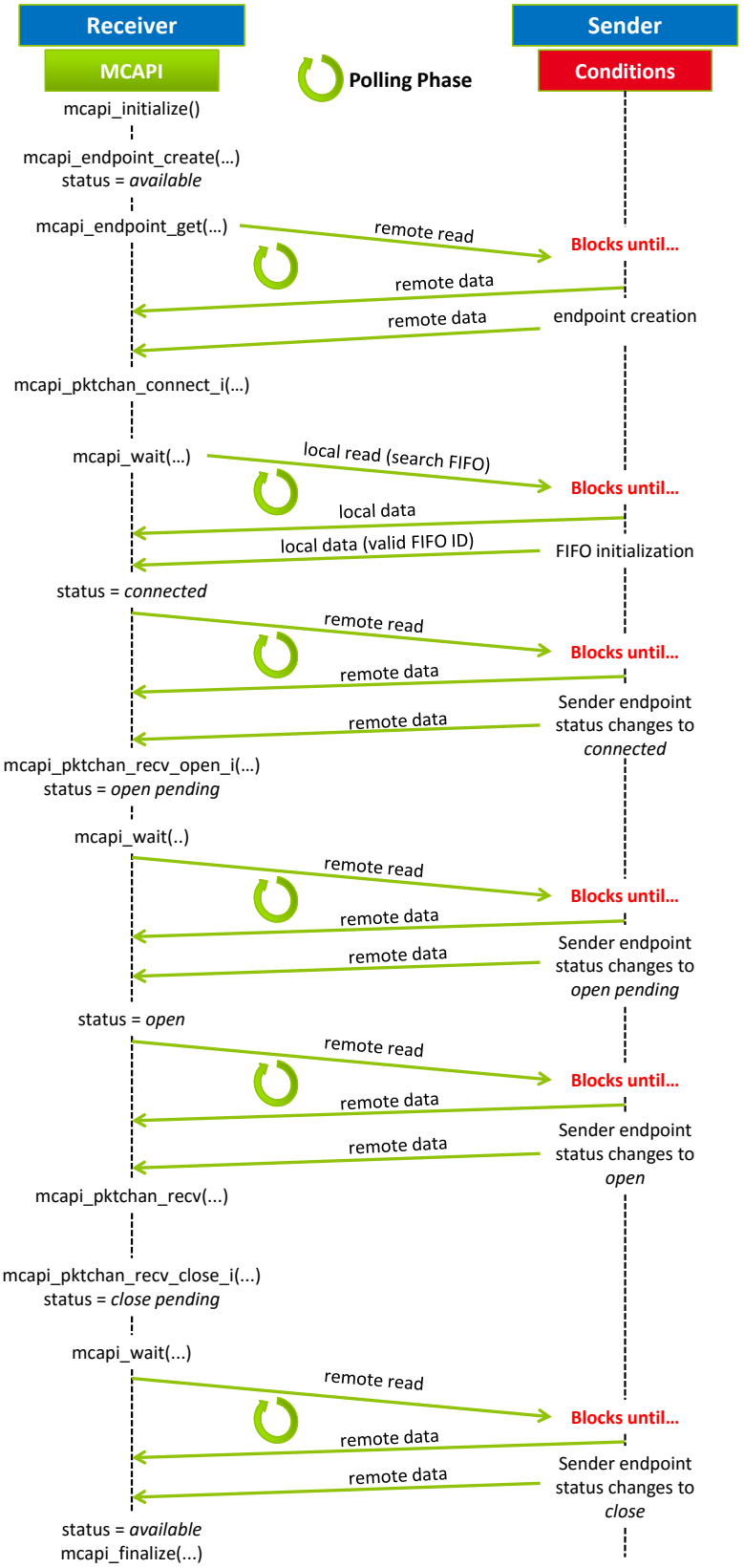


FIGURE 3.2: Connection set-up polling phases in the receiver side.

The additional polling phase occurs due to the *opening* step, where the receiver must assure that the sender side also finishes this step. Otherwise, if the receiver performs the *closing* step right after the `mcapi_wait` operation has completed, the endpoint status will change to CLOSE_PENDING and the `mcapi_wait` operation in the sender side will stall.

The polling phase overhead is twofold: increased network traffic and processing load. The network traffic is increased due to remote reads performed when sender and receiver are not in the same cluster, while the processing load is increased due to execution of several meaningless operations to generate the read packets and check the remote data value. Considering the case where sender and receiver are in the same cluster, there is no traffic overhead. However, the processing load is further increased due to memory reads performed in the local polling phases. Regardless sender and receiver placing, these overheads compromise system performance and efficiency.

Therefore, a flexible hardware module that can handle the different communication set-up steps is required. Additionally, the mechanism must offer support to other functions that present the same behavior, such as the `mcapi_endpoint_get` function. Furthermore, in order to provide further flexibility and to be to be seamlessly compliant with other standard APIs, the software implementation complexity must not be increased.

3.2 Event Synchronizer Mechanism

In order to solve the aforementioned issues, a mechanism called Event Synchronizer (ES) is proposed. This module targets to be as flexible as the solutions presented in [52] and [53], but also aiming co-design with the MCAPI standard. The ES is a programmable hardware module able to handle a parameterizable number of events, which is a software-level defined condition to be accomplished, for each communication terminal in a processing element. Furthermore, each CPU is attached to an independent Event Synchronizer for increased scalability, as showed in the updated cluster representation (Figure 3.3).

Each ES interacts with three modules in the reference architecture cluster: Network Interface (NI), respective CPU and respective Control Registers (Ctrl Regs) (Figure 3.4). The *NI* is responsible for forwarding the synchronization packets to the **ES**. The synchronization packet contains the information of sender and receiver connections, as well as the event code. The *CPU* is responsible for programming the **ES** according to the expected events. The **ES** is responsible for notifying the *Ctrl Regs* module when a

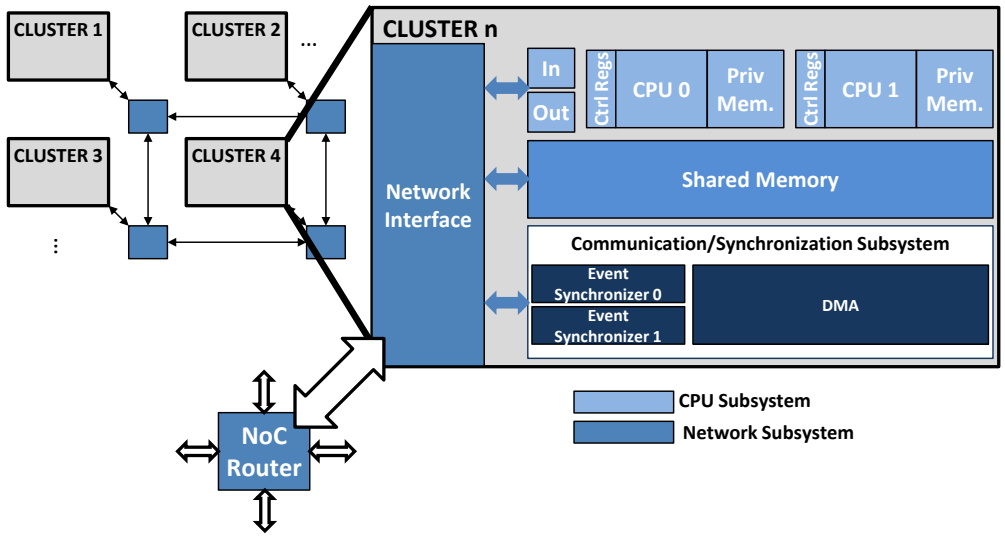


FIGURE 3.3: Cluster block diagram with Event Synchronizer.

received event matches the expected event, or when an expected event matches a stored event. Finally, the *Ctrl Regs* is responsible for generating an interruption for the *CPU*. It is important to note that no reference with MCAPI is assumed, i.e., the ES can be used with any API that uses the concept of termination points to establish a connection. Furthermore, the ES can be used to implement other synchronization actions, such as barriers, by programming the event mask (Figure 3.5) accordingly.

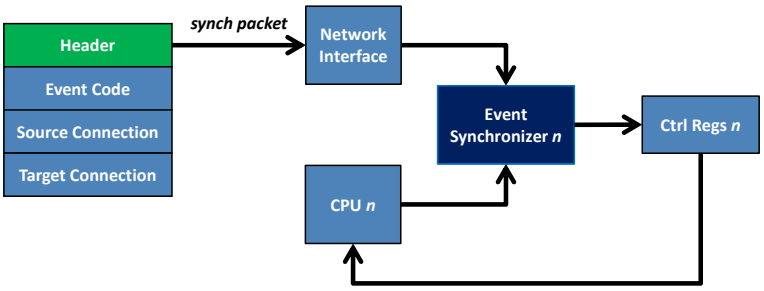


FIGURE 3.4: Synchronization packet and Event Synchronizer block interactions.

The ES hardware structure is showed in Figure 3.5. The ES is composed of multiple Synchronization Event Registers (SERs), Remote Connection ID Registers (RCRs), two mask registers and 4 processes to handle received information and events generation. The SERs are responsible for storing the events for each communication terminal, with its number and size defined by the maximum number of connections and the maximum number of different events that a given processor should handle. In this work, the number of SERs is set to 64 and their size to 12 bits. The RCRs are 32-bit registers used to store the remote termination points identifiers of each connection, which can be used by the software to retrieve the termination point related to the respective connection

or to perform error checking. The number of RCRs is equal to the number of SERs. The mask registers are used by the CPU to program the expected events. The access is performed through memory-mapped registers, where the CPU can write and read the masks. The connection mask defines the termination point to be tested and the event mask defines the expected event. The event mask can also be programmed to expect multiple events, i.e., the ES will notify the *Ctrl Regs* only when a set of events matches the mask value.

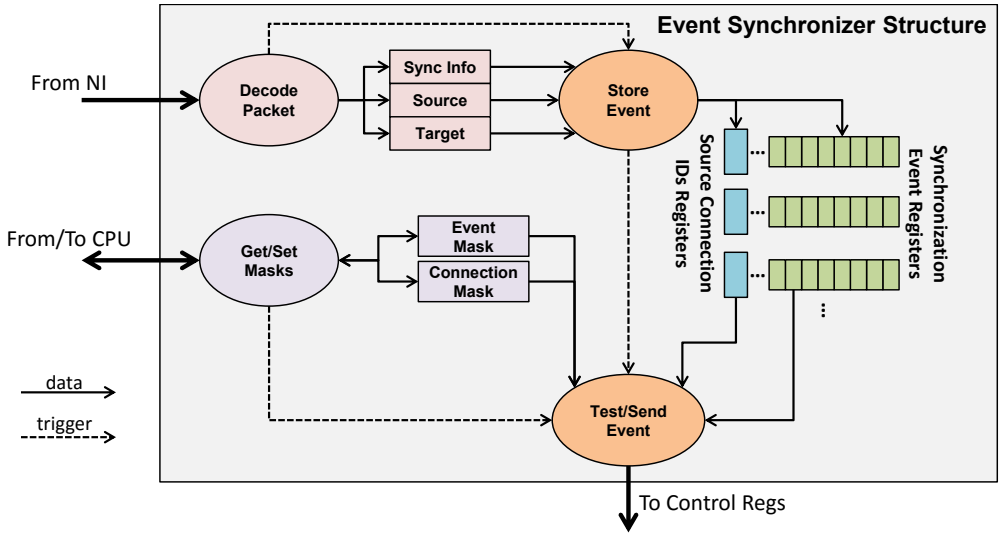


FIGURE 3.5: Event Synchronizer structural view.

Each one of the four processes can be seen as a Finite State Machine (FSM), which wait for data or a trigger signal and execute specific actions. The "*Decode Packet*" process is responsible for receiving the synchronization packet from NI, extracting the information and triggering the "*Store Event*" process. The "*Test/Send Event*" process is responsible for notifying the *Ctrl Regs* when events and masks match. This process is executed in two scenarios: after processing a synchronization packet ("*Store Event*" process) or when masks are updated ("*Get/Set Masks*" process).

Thus, it is possible to remove all polling phases performed during communication set-up in both sender and receiver sides by taking advantage of the Event Synchronizer. This is achieved by defining a different event for each phase of handshake protocol and for each communication set-up step. Figure 3.6 depicts how the ES is used in comparison with the MCAPI software implementation. In addition to avoiding unnecessary network traffic, the ES allows the CPU to enter in "idle" state, which can be translated into a low-power state if this feature is available in the architecture. In this Thesis the "idle" state is considered as a sleep, where the CPU does not execute any instruction and the clock signal is disabled.

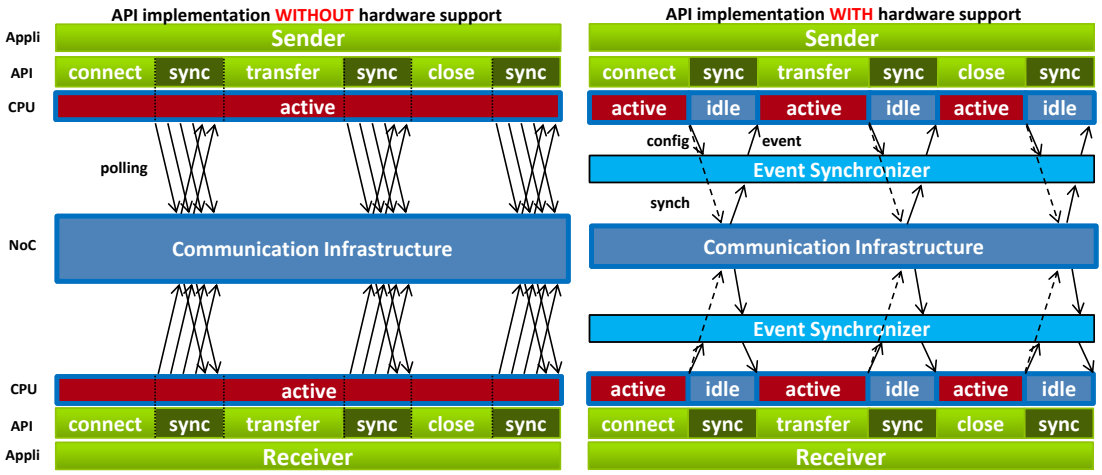


FIGURE 3.6: Comparison of the hardware platform utilization with and without the ES.

An example of the ES detailed operation for the *opening* step is presented in Figure 3.7. In *action 1* the sender open function changes the endpoint status attribute and sends the first synchronization packet to the receiver process. Then, it programs the ES in order to wait for the reception of a synchronization packet signaling that the `mcapi_wait` operation was performed in the receiver process (*action 2*). The *action 3* shows that the synchronization packet arrives in the receiver. However, as the ES was not programmed to be notified for this synchronization packet, the event is stored. In *action 4*, the receiver process programs the ES to wait for the respective event and, as the event was already stored, the CPU is notified. Next, the receiver process sends the second synchronization packet to the sender process and programs the ES to wait for the event representing the end of protocol (*action 5*). Then, *action 6* shows the second synchronization packet arriving in the sender process and the CPU being notified. Finally, the sender process sends the final synchronization packet (*action 7*), which arrives in the receiver process (*action 8*) and, as the receiver was already expecting this event, finishes the *opening* step (*action 9*). The other communication set-up steps are implemented using the same behavior, changing only the ES programming respectively to each function.

3.3 MCAPI Modifications

In order to modify the MCAPI implementation it is considered that each termination point represents one endpoint. Also, as each node (CPU) has its own Event Synchronizer, the only identifier used to correlate a SER and an endpoint is the *port_id*. Finally, nine events were defined in order to differentiate the communication set-up steps (Table 3.1).

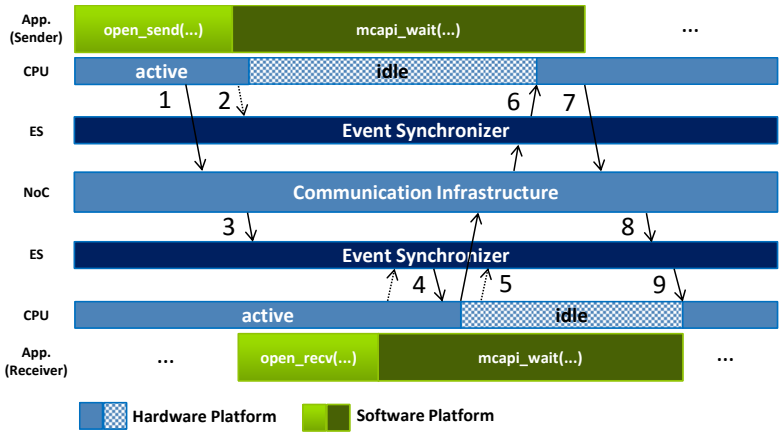


FIGURE 3.7: Communication opening diagram using the Event Synchronizer.

TABLE 3.1: MCAPI communication set-up events.

Event	Code
SENDER_INIT_FIFO	0
SENDER_CONNECTED	1
RECEIVER_CONNECTED	2
SENDER_OPEN_PENDING	3
RECEIVER_OPEN_PENDING	4
SENDER_OPEN	5
RECEIVER_OPEN	6
SENDER_CLOSE	7
RECEIVER_CLOSE	8

These events can cover all the connection set-up steps and are mainly used in the `mcapi_wait` operation. Furthermore, two functions were created to avoid code replication: `mcapi_trans_wait_synch` and `mcapi_trans_send_synch`. These functions are called when waiting or sending a given event and are presented in Figures 3.8 and 3.9, respectively. Both functions receive as parameters the remote and local endpoint identifiers and the event code (`info`) to be sent or expected. The *wait_synch* function programs the ES and set the CPU to the idle state, while the *send_synch* function sends the synchronization packet. These functions are used in the implementation of the primitives employed in the connection set-up steps and in the implementation of the `mcapi_wait` operation.

Figure 3.10 presents a piece of code containing the implementation of the `mcapi_wait` operation for the `mcapi_pktchan_send_open_i` function. The programmer can define to use or not the Event Synchronizer by defining the `POLLING_SET_UP` directive. As it can be seen, if the directive is defined, the compiler will use the polling code, where the receiver endpoint status is read until the status changes to *connected*. Alternatively,

without the directive definition, the implementation will take advantage of the ES by using the *wait_synch* and *send_synch* functions. In this specific example, these functions represent *action 2* and *action 7* in Figure 3.7.

Therefore, thanks to hardware and software co-design, the bottlenecks of the communication set-up phase were identified and transfered to a hardware module. As result, the mechanism is able to handle multiple steps of the MCAPI implementation and showed to be easily accessible and programmable. Moreover, the programmer does not need to change the application code since the software modifications are performed only at the API level. Finally, the performance evaluation and the overhead in terms of code size is presented in Chapter 5, showing that the ES can significantly decrease network and processor loads with minimal increase in memory footprint.

```
mcapi_boolean_t mcapi_trans_wait_synch(
        mcapi_endpoint_t local_ep,
        mcapi_endpoint_t remote_ep,
        mcapi_uint32_t info)
{
        unsigned int mips_id;

        // Check if the connected endpoint is a valid endpoint.
        if(mcapi_trans_valid_endpoint(remote_ep))
                return MCAPI_FALSE;

        // Retrieves the local node identifier.
        mcapi_trans_get_node_num(&mips_id);

        // Set the IT Control module to interrupt the CPU when
        // the Event Synchronizer signals that an event has arrived.
        set_mask_sleep(SLEEP_MASK_ALL);
        unset_mask_sleep(IT_PKT_SYNCH);
        clear_status(IT_PKT_SYNCH);

        // Programs the Event Synchronizer with the respective
        // endpoint identifier and event code.
        set_mask_synch_conn(mips_id, local_ep);
        set_mask_synch_info(mips_id, (1<<info));

        // Reads the status of the CPU and set the sleeping mode
        // in case no interruption has occurred.
        if(!read_status_sleep())
                sleeping_mode();

        // CPU Pause -> Next instruction executed only when the CPU is
        // woken up

        // Clear Event Synchronizer Masks.
        clear_mask_synch_info(mips_id,(1<<info));
        clear_mask_synch_conn(mips_id);

        return MCAPI_TRUE;
}
```

FIGURE 3.8: Source code of the wait_synch function.

```
void mcapi_trans_send_synch(
  mcapi_endpoint_t remote_ep,
  mcapi_endpoint_t local_ep,
  mcapi_uint32_t info)
{
  mcapi_uint32_t remote_domain, remote_node;

  // Decodes domain and node identifiers for the remote endpoint.
  mcapi_trans_decode_endpoint(remote_ep, &remote_domain, &remote_node,
    NULL);

  // Uses the lower level software function to send the packet.
  // This function sends the source, target and info variables
  // in the NI and informs it is a synchronization packet.
  com_api_send_synch_packet(
        remote_domain,
        remote_node,
        local_ep,
        remote_ep,
        info);
}
```

FIGURE 3.9: Source code of the send_synch function.

```
#ifdef POLLING_SET_UP
        timeout_count = 0;
        while(!mcapi_trans_endpoint_channel_isopen(receiver_endpoint)){
                timeout_count++;
                if(timeout_count == timeout){
                        *mcapi_status = MCAPI_TIMEOUT;
                        return MCAPI_FALSE;
                }
        }
#else
        // The CPU is set to sleeping mode until receive the
        // RECEIVER_OPEN event.
        if (!mcapi_trans_wait_synch(receiver_endpoint, sender_endpoint,
    RECEIVER_OPEN))
                return MCAPI_FALSE;

        // Sends the SENDER_OPEN event.
        mcapi_trans_send_synch(sender_endpoint, receiver_endpoint,
    SENDER_OPEN);

#endif
```

FIGURE 3.10: Implementation of polling-based and event-based approaches for the opening step in the sender side.

Chapter 4

Data Transfer Support

This chapter presents the third contribution of the Thesis, which is the development of a hardware module to decrease computation and traffic overheads described in Section 2.3.2. This module is responsible for managing buffers used in the data transfer and for packing/unpacking the sent/received data. The module is called Buffer Manager and, similarly to the Event Synchronizer (Chapter 3), was developed in co-design with MCAPI, targeting flexibility and low programming complexity. In order to introduce the issues addressed by the proposed module, the data transfer phase and the software implementation of FIFO control are reviewed in Section 4.1. Then, Section 4.2 details the Buffer Manager implementation. Finally, the modifications performed in the MCAPI implementation to take advantage of the Buffer Manager are presented in Section 4.3.

4.1 Data Transfer Phase

The data transfer phase is performed once the communication set-up is completed and the channel is opened. The MCAPI implementation uses software FIFOs mapped in the shared memory (Section 2.2.1.2) to send and receive data. Furthermore, the implementation uses the functions provided by the reference architecture software stack (Section 1.4) to avoid managing and translating addresses, which decreases the implementation complexity. Thus, the `mcapi_pktchan_send` and `mcapi_pktchan_recv` functions use an additional software layer, as depicted in Figure 4.1.

The *FIFO API* is responsible for interfacing the MCAPI implementation and the hardware blocks. On the sender side, two functions are available: `fifo_write` and `fifo_write_block`. These functions are able to send a 32-bits value and a buffer of 32-bit values, respectively. On the receiver side, the only available function is the

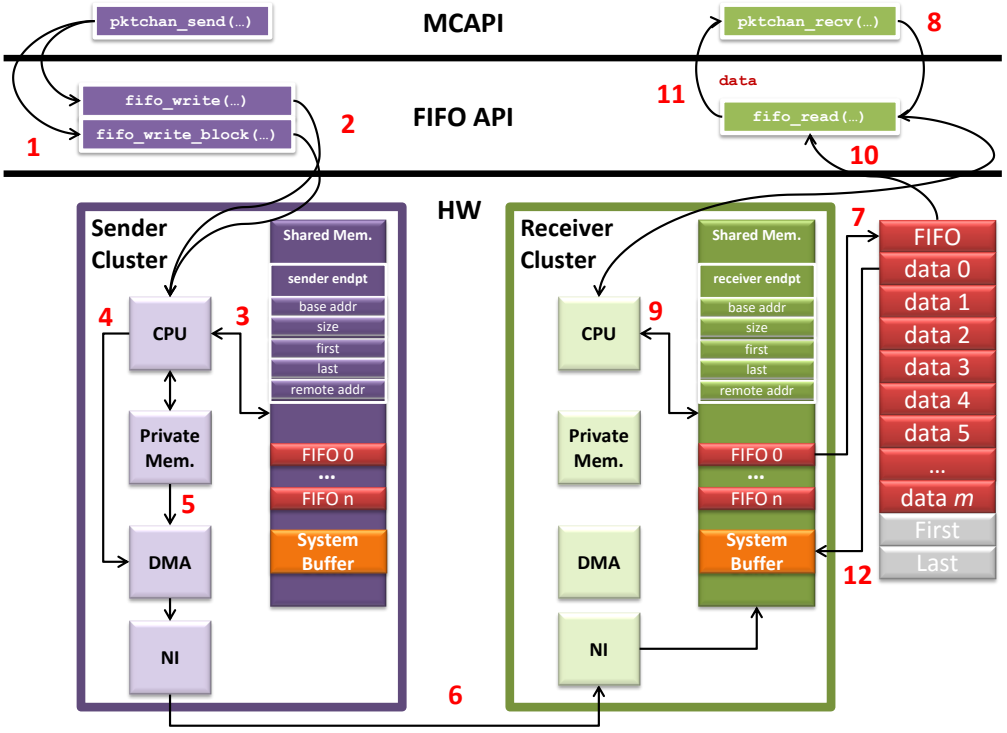


FIGURE 4.1: Steps performed during data transfer using pure software implementation.

`fifo_read`, which returns a 32-bit value. The numbers showed in Figure 4.1 represent the order of the actions when sending and receiving data, up from MCAPI implementation. The actions performed when sending data with the `fifo_write_block` function are represented by the numbers from 1 to 7 (up to writing the data in the destination FIFO), while the numbers from 8 to 12 represent the actions performed in the receiver side.

The first action executed by the `mcapi_pktchan_send` function is to check the amount of data being sent and call the respective write function. The `fifo_write_block` function is used when sending more than one 32-bit value (word). This function presents lower traffic overhead since it uses the DMA block to perform read/write operations and updates the FIFO write pointer only once. On the other hand, it presents higher processing overhead, due to the address control needed when updating the write pointer (wrapping), and is avoided when transferring only one word. Next, the FIFO descriptor attribute (Section 2.2) is accessed in order to retrieve the destination address (*action 3*), followed by the DMA request creation (*action 4*). Then, the DMA reads data from memory (*action 5*), packs and sends it to the remote Shared Memory through the NI (*action 6*). In this particular example, the data is read from the Private Memory. However, the DMA is also able to read data from the Shared Memory. Finally, in the receiver cluster, *action 7* represents the data being written in the respective FIFO.

The process is very similar when using the `fifo_write` function. The main difference is that the DMA is not used, since the data is supplied by the application, i.e., it is already in the CPU. Thus, the *action 5* is skipped and the *action 4* directly creates the packet to the NI, which, in turn, follows the same behavior.

In the receiver side, the `mcapi_pktchan_recv` take advantage of the `fifo_read` function (*action 8*). The FIFO descriptor is accessed (*action 9*) and the data is returned from the FIFO (*action 10*) to the MCAPI implementation (*action 11*). Finally, the MCAPI implementation moves the data to the System Buffer (*action 12*) and returns its address. However, contrary to the sender side, the read function does not take advantage of the DMA block. Thus, when receiving a buffer, each word is read and copied sequentially, impacting communication performance. Additionally, although the pointer updates are not depicted in Figure 4.1, it creates additional traffic overhead, which also impacts communication performance (Section 2.3.2). This overhead is induced by the packets sent to the remote cluster after executing every write or read function, as explained in Figure 2.9.

Therefore, in order to completely decouple computation and communication and to increase system performance, a hardware mechanism to manage buffers/FIFOs should be exploited. Also, similarly to the Event Synchronizer, the mechanism must be flexible and easily programmable in order to avoid an increase in the software implementation complexity and to be seamlessly compliant with other standard APIs.

Figure 4.2 illustrates how the mechanism could work: two hardware blocks (Writer and Reader) are introduced to manage the data access in the shared FIFO buffer. These blocks detach the low level communication management from Sender and Receiver processes and could be accessed and/or programmed through memory-mapped registers. Three partitioning schemes, depicted by the numbers 1, 2 and 3, are considered. Each number represents a partitioning of blocks among sending and receiving clusters: left side of the dotted line is implemented in the sender side, while the right side is implemented in the receiver side.

Partitioning 1 and 3 are equivalent since the FIFO buffer is exclusively placed in one of the clusters with the respective blocks for writing and reading data. On the other hand, the partitioning 2 completely separates the writing and read blocks in their respective clusters. In common, all schemes alleviate the processing overhead due to the FIFO buffer control being performed in hardware. However, the partitioning 2 presents a higher overhead in network traffic since the Writer and Reader blocks must exchange the pointer addresses. Furthermore, the main advantage partitioning 1 has over partitioning 3 is to require remote writes instead of remote reads when transferring data, presenting a lower latency in the reference architecture, as mentioned in Section 2.2.1.2. Thus,

the communication mechanism presented in the next section is implemented using the partitioning scheme 1.

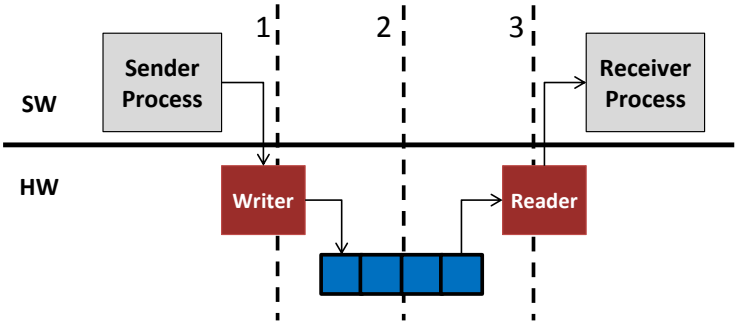


FIGURE 4.2: Partitioning schemes options to implement a buffer management solution in hardware.

4.2 Buffer Manager Mechanism

The proposed mechanism is called Buffer Manager Mechanism (BMM) and targets to decrease the computation and traffic overheads in inter-process communication. The main objectives to achieve this are:

- Accelerate the FIFO buffer management by using hardware implementation;
- Decrease control complexity by avoiding pointer exchange;
  - Implementation of an end-to-end credit flow control mechanism;
- Abstract addresses by using port identifiers (IDs) in the communication;
- Increase flexibility by providing three different communication modes:
  - Address-based transfer – from a source to a destination address (DMA-like).
  - Direct data transfer – a single 32-bit word from source ID to destination ID.
  - Buffer transfer – a buffer of variable size from source ID to destination ID.

The BMM is composed of four hardware modules, as depicted in Figure 4.3. This mechanism was developed in co-design with MCAPI and replaces the DMA in the Communication and Synchronization Subsystem (Section 1.4) to handle data transfers. The main differences between the BMM and the DMA are that the earlier can handle read operations as well as write operations, and the use of port identifiers to implement a connection-based communication for direct data and buffer transfers. Also, due to the port identifiers, the BMM is able to handle multiple connections in parallel through read and write requests, reducing the hardware cost while increasing flexibility.

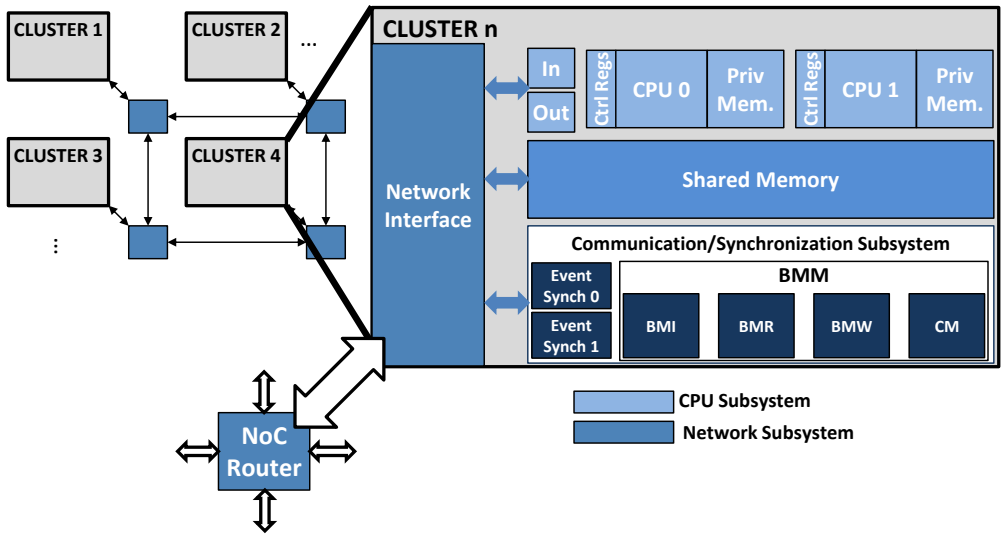


FIGURE 4.3: Cluster block diagram with BMM modules.

The Buffer Manager Interface (BMI) is responsible for fetching and packing the data to be transmitted in the sender side, while the Buffer Manager Write (BMW) is responsible for, in the receiver side, receiving and writing the data in the respective buffer. The Buffer Manager Read (BMR) is used in the receiver side to fetch the received data from the buffer. Finally, the Credit Manager (CM) implements the communication flow control through a credit-based policy. The CM is responsible for sending and updating credits when data is read or sent, respectively. Additionally to the hardware modules, the BMM uses three table structures to control the communication: Connection Table, Credit Table and Buffer Table. These tables are implemented as registers, while the buffers are still placed in the cluster Shared Memory. However, since the mechanism is programmable, the tables could be placed in a memory and buffers could be implemented as a hardware block, for instance. The implementation of these blocks are detailed in the next sections.

Drawing a parallel with Figure 4.2, the BMM blocks are organized as depicted in Figure 4.4 (the Credit Manager block is omitted since the figure represents only the data flow during the communication). Using this approach, the FIFO buffer is managed only in the receiver side, avoiding pointers exchanging between sender and receiver. Additionally, from the software point of view, there is no need to manage remote addresses since the sender processes use only connection IDs, thus, simplifying the API programmability.

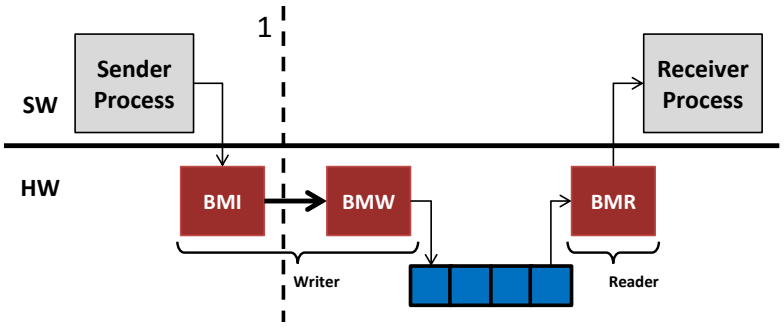


FIGURE 4.4: Partitioning of BMM blocks regarding communication sides.

4.2.1 Table Structures

The table structures are responsible for storing important parameters in the communication process: connection identifiers, available credits and buffer descriptors. These tables are implemented as registers and are accessed by the CPU as memory-mapped registers. The access to read and write from/in the tables is performed by using the primitives `read_table(table, id)` and `write_table(table, id, data)`, where `table` and `id` represent the table (each structure has a single identifier) and the port identifier to be accessed, respectively. This implementation allows the software API to seamlessly access the table structures without significant performance degradation, since read and write operations are performed similarly as in the Private Memory.

The Connection Table (CT) stores the port identifiers (ID) that are connected to each local port. Each position of the CT refers to the local port ID, e.g. position 0 refers to local port 0, position 1 to the local port 1, and so forth. Thus, the size of the CT depends on the number of maximum ports per process/CPU. As each port can be used as input or output, the value stored in the CT may refer to destination or source ID. If a given port is being used as output port, the value stored in its respective position of the CT informs the ID of the remote input port (destination ID). On the other hand, if a given port is being used as input port, the value stored in its respective position of the CT informs the ID of the remote output port (source ID). It is a programmer responsibility to manage the direction and configure the port IDs during the connection set-up phase (Section 4.2.2).

The Credit Table (CrT) stores the available credits for each output port of the CPU. Each entry corresponds to a local port ID. If a given port is not being used or is being used as input, the value of its respective entry is zero. Since all the ports in a CPU can be used as output ports, the number of entries in the CrT is the maximum number of ports in a process/CPU. As the number of credits represents the buffer available space in bytes, the width in bits of each position depends on the buffer size (e.g. for a buffer

size of 128 bytes, each entry would be an 8-bits register). This table is written only by the Credit Manager module. Thus, attempting to write in this table from the CPU results in an error.

The Buffer Table (BT) stores the buffer descriptors for each local port ID. In order to simplify the design, the BT size is the maximum number of ports in a process/CPU. Therefore, if a given port ID does not have an associated connection, its entry in the BT is `NULL`. Each BT entry stores the buffer base address in the Shared Memory, buffer size, read and write pointers and a credit threshold. This table is written by the CPU only during connection set-up. During the data transfer, it is BMW and BMR responsibility to update the write and read pointers, respectively.

4.2.2 Connection Set-up

Although only the data transfer phase takes advantage of the BMM, few initialization steps are required during the *opening* step of communication set-up. These steps are called connection set-up and must be accomplished by both communicating ports, as further detailed in Section 4.3. As each port can be used as output (sender) or input (receiver), the connection, credit and buffer table structures must be filled accordingly in both sides. Therefore, the following steps must be performed to set-up the connection:

From Sender side:

– The CPU updates the Connection Table according to the input and output port IDs;

From Receiver side:

– The CPU updates the Connection Table according to the input and output port IDs;

– The CPU updates the Buffer Table with the buffer descriptor according to the input port ID;

– The CPU creates the Initial Credit Packet (Table 4.4), sending credits to the process related to the output port ID;

Finally, when the set-up phase is finished, the CPUs can start sending and receiving data by creating requests to the Buffer Manager Interface (BMI) in the sender side and to the Buffer Manager Read (BMR) in the receiver side. The BMI, BMR and Table

Structures are mapped in the Global Address Map as showed in Table 4.1. The bits 22, 21 and 20 defines that the CPU or peripherals are accessing the Buffer Manager and Tables. In case of accessing the BMM, the bit 19 defines if the data will be delivered to the BMI or to the BMR. The request creation and the parameters (bits 18 to 0) are detailed in the next section.

TABLE 4.1: Buffer Manager mapping into the Global Address Map.

Cluster ID	Group	Address					Module
31 23		22	21	20	19	18 0	
N	RAM	0	0	...		...	Private and Shared Memories.
	Peripherals	0	1	0	0	...	Status Registers,
		0	1	0	1	...	Input, Ouput, etc.
	Buffer	**0**	**1**	**1**	**0**	**Parameters**	**Buffer Manager Interface.**
	Manager	**0**	**1**	**1**	**1**	**Parameters**	**Buffer Manager Read.**
	Tables	*1*	*0*	*1*	*Parameters*		*Connection, Credit and Buffer Tables.*
	Unused	1	1	...		...	

4.2.3 Data Transfer Requests

The CPU schedules send and receive operations by creating requests to the BMM. To create a request, the data has to be written in specific addresses that encode the request parameters, referred as configuration address. When sending data, three options are available: (*i*) address-based transfer, (*ii*) direct stream-based transfer, or (*iii*) indirect stream-based transfer. The address-based request is implemented as a legacy functionality and used to perform transfers previously addressed to the DMA, where the source and destination addresses are explicitly provided. On the other hand, the stream-based requests abstracts the destination addresses through the port IDs. In the direct stream-based request a single word of 32-bits is transmitted from a source port ID to a target port ID, while in the indirect stream-based request a buffer of 32-bits words is transmitted from a source port ID to a target port ID. As the BMM is used to accelerate FIFO-like data transfers, only options (*ii*) and (*iii*) are supported when receiving data. Thus, when the address-based transfer (option *i*) is used to send data, the receiver process must know the destination address and read the data using the functions provided by the HAL layer.

The number of writes required to generate a request varies according to its type. The address-based transfer request requires three writes, while word and buffer transfer requests require one and two writes, respectively. Table 4.2 shows the request fields coded in the 18 LSBs of the configuration address, and the selection of BMI and BMR memory-mapped registers from bits 22 to 19. The main objective of encoding the request parameters in the configuration address is to decrease the number of memory writes needed to complete a request, hence, lowering the software processing overhead in the overall communication performance.

TABLE 4.2: Buffer Manager request parameters encoding.

31 23	22	21	20	19	18 17	16	15 8	7 4	3 0
CLUSTER ID	**0**	**1**	**1**	**R/W**	**T**	**EoR**	**port/src id/flag**	**CPU ID**	**Unused**
R/W - Read or Write operation; T - Type; EoR - End of Request ;									

The bit 19 defines the request operation: Write for BMI or Read for BMR. The bit 18 defines request Type (address-based, indirect or direct stream based) and the bit 16 signals that the request has been completed (End of request). The port, src id or flag information are encoded from the bit 15 to 8, with the respective information depending on the request type. Finally, the CPU ID is coded from the bit 7 to 4. The 4 LSBs (3 to 0) are left unused.

4.2.3.1 Write Request

The write request is defined by the value "0" in the bit 19 of the configuration address and is directed to the BMI. The other bits are filled according to the request type, CPU and port/src id/flag. Table 4.3 details the possible configuration options when creating a write request. The CPU ID information is available at the HAL layer after the system boot, and thus, does not need to be provided at every request write. Instead, when the CPU writes the data for the BMM, it encodes this information accordingly.

When the request type is the address-based transfer, the bit 18 is set to '0' and the bits 17 and 16 are changed according to the parameter to be informed. Three writes are needed to complete an address-based transfer request. The first write will inform the "Source Buffer Address" (line 1) and the second one the "Target Buffer Address" (line 3). The third and last write must inform the "Buffer Size" (line 4), which completes the request by setting the bit 16 (EoR) to '1'. Additionally to the data, the `SRC ID` parameter is coded from the bits 15 to 8 in the second write. This parameter is needed to create the packet header that is sent through the NI.

TABLE 4.3: Write requests encoding.

	Request Type	Written Data	R/W	Type		EoR		
			19	18	17	16	15 8	7 4
1	Address Based Transfer	Source Buffer Address	**0**	0	0	0	-	CPU ID
2	Address Based Transfer	Not used	0	0	0	1	-	-
3	Address Based Transfer	Target Buffer Address	**0**	0	1	0	SRC ID	CPU ID
4	Address Based Transfer	Buffer Size	**0**	0	1	1	-	CPU ID
5	Stream Based Direct (data)	Not used	0	1	0	0	-	-
6	Stream Based Direct (data)	Data to be transmitted	**0**	1	0	1	Port ID	CPU ID
7	Stream Based Indirect (buffer)	Source Buffer Address	**0**	1	1	0	Port ID	CPU ID
8	Stream Based Indirect (buffer)	Buffer Size	**0**	1	1	1	*Flag*	CPU ID

When using the stream-based transfers, the bit 18 is set to '1'. In this transfer type two request types are possible: Direct and Indirect. In the direct transfer (line 6) the bit 17 is set to '0' and the bit 16 to '1' since only one write is needed (informing the data to be transferred). On the other hand, the indirect transfer takes two writes, informing the source buffer address (line 7) and the buffer size (line 8). In both request types the sender port ID is coded from the bits 15 to 8, since the BMI needs this information to retrieve the respective destination port ID in the Connection Table. Furthermore, for indirect stream-based transfers the `flag` parameter might be informed. When this parameter is specified (any value other than zero) and the transfer is finished, the BMI sends a loop-back synchronization packet to the respective Event Synchronizer (based on the CPU ID) with the informed flag as the event code, signaling that the source buffer can be reused.

4.2.3.2 Read Request

The read request is defined by the value "1" in the bit 19 of the configuration address and is directed to the BMR. The other bits are filled according to the request type, CPU and port IDs. Table 4.4 details the possible configuration options when creating a read request. Contrary to write requests, only the stream based requests are available. This is because the application uses the CPU to read directly from specific memory addresses. Thus, there is no need to provide support for address-based requests when reading data. Consequently, the bit 18 is always set to '1' when creating a read request.

The direct data transfer has a different behavior when compared to the other requests. In this case, the CPU reads from the configuration address instead of writing on it. Thus, the CPU stays blocked until the BMR replies with the read data, the same way that when reading from memories. The information coded from the bits 15 to 8 is

TABLE 4.4: Read requests encoding.

	Request Type	Written Data	R/W	Type		EoR		
			19	18	17	16	15 8	7 4
1	Initial Credit Generation	Number of Credits	1	0	0	0	Port ID	CPU ID
2	Address Based Transfer	Not used	1	0	0	1	-	-
3	Address Based Transfer	Not used	1	0	1	0	-	-
4	Address Based Transfer	Not used	1	0	1	1	-	-
5	Stream Based Direct (data)	Not used	1	1	0	0	-	-
6	Stream Based Direct (data)	(read access)	**1**	1	0	1	Port ID	CPU ID
7	Stream Based Indirect (buffer)	Destination Buffer Address	**1**	1	1	0	Port ID	CPU ID
8	Stream Based Indirect (buffer)	Buffer Size	**1**	1	1	1	*Flag*	CPU ID

the receiver port ID, which is used to retrieve the respective source port ID and buffer descriptor.

When using the indirect transfer, the bit 17 is set to '1' and two writes are needed. The first write (with bit 16 set to '0') informs the destination address and encodes the receiver port ID. The second write (bit 16 set to '1') informs the buffer size to be read and might encode a `flag` parameter, which can be used in a similar way as in the write requests, to inform that the destination buffer has been filled.

Besides the read requests, the BMR is also used in the communication set-up. Since the Credit Manager is not directly accessible by the CPU, an unused configuration address was selected to encode the generation of the initial credit packet (Section 4.2.2). Thus, when the initialization is performed, the amount of credits is written in the corresponding configuration address, which also encodes the local port ID (Table 4.4 - line 1). As the Connection Table is initialized prior to credit generation, the BMR is able to retrieve the corresponding port ID and signal the Credit Manager to generate the credit packet.

4.2.4 Data Transfer Operations

Figure 4.5 depicts how the mechanism works during an indirect stream-based request in both communication sides. Firstly, there is a set-up phase in both sides (*action 1*), which consists of the CPU initializing the Connection Table with the remote port IDs (R ID) that will be sending/receiving data for the respective local ports (L ID). Although it is not represented for clarity purposes, in the receiver side, the CPU is also responsible for initializing the Buffer Table and requesting the initial credit generation to the BMR. Next, in the Sender cluster, the CPU writes data in a buffer in the local memory and creates a write request for the BMI (*action 2*). Then, the BMI picks the

request and retrieves the remote port ID and available credit for the local port ID from the Connection Table and Credit Manager, respectively (*action 3*). In case of available credits, the data packet is sent through the NoC (*action 4*) with the information of the target port ID, while the CM is notified to update credits for the respective local port ID. Otherwise, the request is not dequeued and the BMI searches for other requests.

In the Receiver cluster, the data is received by the BMW, which accesses the Buffer Table (*action 5*) based on the target port ID and writes the data in the respective buffer (*action 6*). At a given moment, the CPU in the Receiver cluster creates a read request for the BMR (*action 7*). Then, the BMR access the Buffer Table (*action 8*) based in the local port ID and copies the data from the respective buffer to the target address (*action 9*). Finally, the BMR notifies the CM to generate credits to the remote port ID (*action 10*), which, in turn, creates and sends the credit packet through the NI (*action 11*).

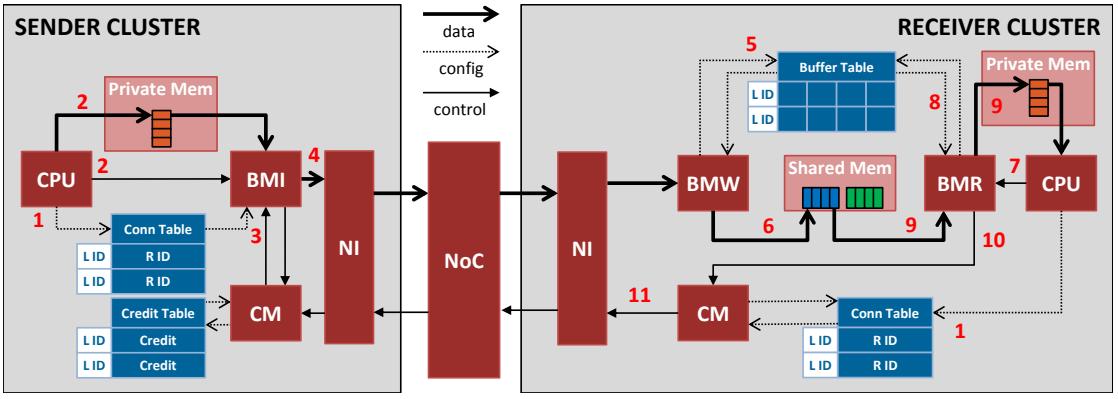


FIGURE 4.5: Buffer Manager Mechanism operation in an indirect stream-based request.

For direct stream-based requests the process is very similar from the hardware point of view in both communication sides. However, as only one 32-bit word is transmitted, the software does not pass a buffer address to be read or filled, but provides the data directly to the BMM. Also, the number of credits needed to complete the request corresponds to 4 bytes. Finally, in address-based transfers, the BMI does not need to retrieve information from Tables and Credit Manager, but should pack the data with the respective destination address before handling the packet to the NI.

4.2.5 Buffer Manager Interface (BMI)

The Buffer Manager Interface is the block responsible for receiving send requests from the CPUs (software layer) and performing the data transfers in the sender side. The BMI is connected to the CPU, Memories, Credit Manager (CM), Connection Table

and Network Interface (NI), as depicted in 4.6. The CPUs write the requests for the BMI through memory-mapped registers, as detailed in Section 4.2.3.1. The data is fetched from Private or Shared Memory, depending on the provided source address. Credit Manager and Connection Table are accessed to retrieve the available credit and destination port ID information, respectively. Finally, the Network Interface is used to send the data and the Credit Manager to update the available credits for the respective port.

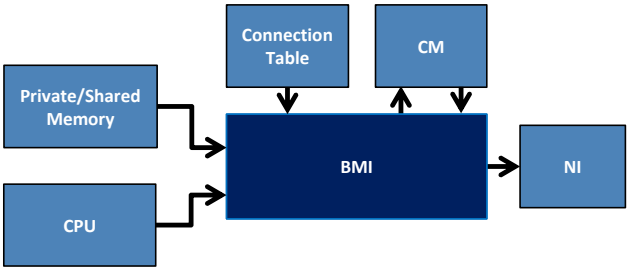


FIGURE 4.6: Buffer Manager Interface connected modules.

The BMI function view is presented in Figure 4.7. The BMI is implemented in 4 main processes, which can be translated into Finite State Machines (FSMs), and has CPU-independent queues, which store the send requests and are managed as hardware FIFOs. The *Request Decode* is the first process is and receive as input the data written by the CPU and the configuration address. This process decodes the information from configuration address and assembles them together with the written data into a request. Then, the `CPU ID` information retrieved from the configuration address is used to store the send request in the respective queue. An alternative implementation is to connect each CPU directly to its respective queue, allowing multiple CPUs to store send requests in parallel. However, this implementation might significantly increase area and power overheads as the CPU count increases.

Next, the requests are selected by the process *Request Selection & Credit Test*. This process selects the requests using a round-robin policy. However, for stream-based transfers, the request is selected only if there are available credits for the data transfer. Thus, the send requests must have the respective buffer size as maximum transfer size in order to be successfully handled. When the application has to perform data transfers larger than the buffer size, it must create multiple requests of smaller size. This constraint avoids data transfer deadlocks that might be caused due to the request scheduling and allows other data transfer flows to start when a given receiver buffer is full. The available credits information is retrieved through the Credit Manager block.

Then, after request selection, the process *Request Processing* is triggered. This process is responsible for retrieving the destination ID from the Connection Table and

for triggering the *Data Fetch and Packing* process according to the request type. In the *Data Fetch and Packing* process, the data is read from the respective source address and packed according to the request type. For address-based transfers, the destination address is packed after the packet header, while in stream-based transfer, the destination port ID is packed within the packet header.

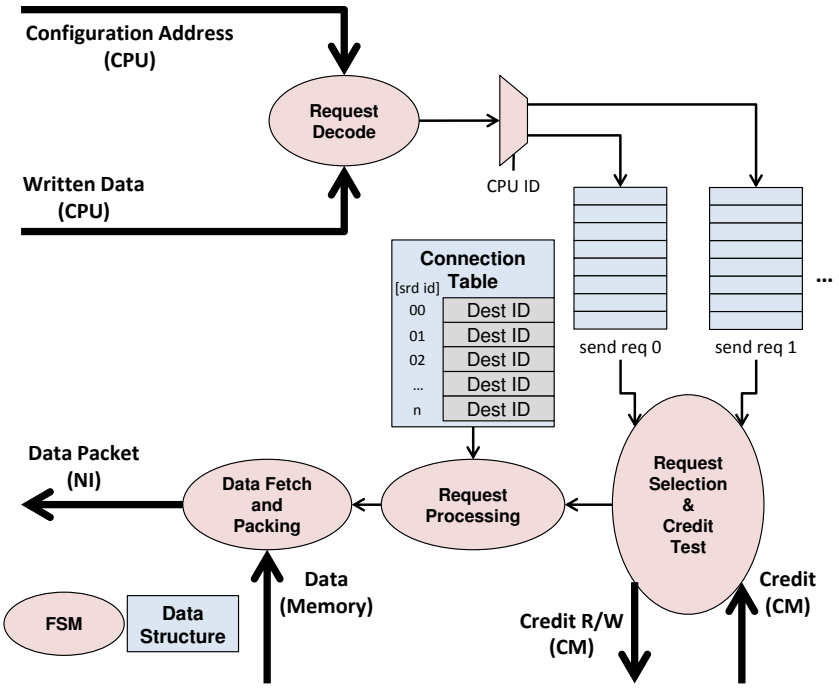


FIGURE 4.7: Buffer Manager Interface functional description.

4.2.6 Buffer Manager Write (BMW)

The Buffer Manager Write is the smallest BMM module. Indeed, it can be considered a complement for the BMI placed in the receiver process, as discussed in Figures 4.2 and 4.4. It is connected to the Network Interface (NI), Buffer Table and Shared Memory, as showed in Figure 4.8. The NI directs the packet to the BMW when it identifies a stream-based data packet. The Buffer Table is accessed to retrieve the buffer descriptor, allowing the BMW to write in the respective destination buffer. The connection with the Shared Memory is related to the implementation realized by this Thesis, which places the buffers in the Shared Memory and implement them as software FIFOs. However, in cases where the buffers are mapped in other structures or are implemented in a different way, the BMW must access the respective location, as well as the information/addresses stored in the buffer descriptor.

Figure 4.9 presents the BMW functional view. As the only source of data is the NI, which receives one packet at a time, the BMM has a single process (*Data Processing*).

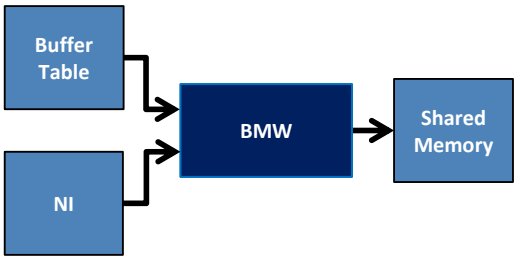


FIGURE 4.8: Buffer Manager Write connected modules.

This process is triggered by the reception of a new packet from the NI. The first action is to retrieve the buffer descriptor from the Buffer Table. Next, the data received from NI is written in the respective buffer. At the same time, the BMW updates the buffer write pointer of the respective buffer descriptor in the Buffer Table. These actions are executed until the entire data packet is received. Then, the *Data Processing* process starts to wait for a new package. Finally, tt is important to highlight that, due to the credit-based flow control, the BMW always has available space to write the data in the destination buffer, which avoids NoC contention and possible deadlocks.

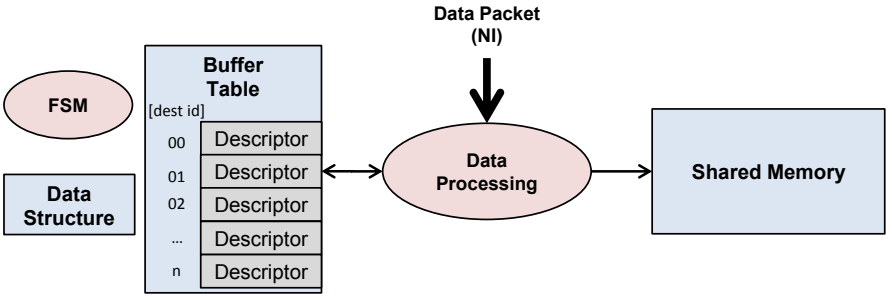


FIGURE 4.9: Buffer Manager Write functional description.

4.2.7 Buffer Manager Read (BMR)

The Buffer Manager Read module is responsible for storing the read requests created by the CPU (software layer) and performing the data transfers in the receiver side. The BMR interacts with the Credit Manager (CM), Buffer Table, Shared and Private Memories and the CPU (Figure 4.10). Similarly to send requests in BMI, the read requests are created by writing data in memory-mapped registers and stored in CPU-independent queues (hardware FIFOs). As detailed in Section 4.2.3.2, the destination address must be informed as one request field, and can belong either to Private or Shared Memory. The Buffer Table is accessed to retrieve the respective buffer descriptor that stores the read pointer address. As mentioned in Section 4.2.6, the reason to access the

Shared Memory to retrieve the received data is due to the FIFO buffers being mapped there.

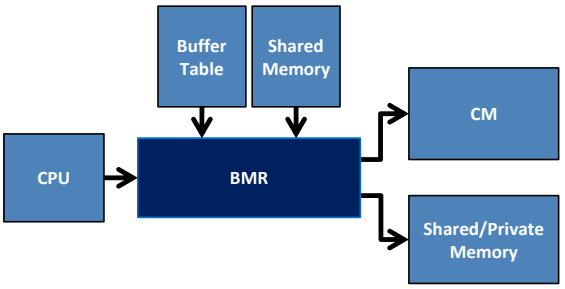


FIGURE 4.10: Buffer Manager Read connected modules.

The BMR functional behavior is detailed in Figure 4.11. The left side shows the *Request Decode* process, which is responsible for decoding the written data and configuration address generated by the CPU. After the request is completed (EoR bit set to '1'), the request type is evaluated to decide which process have to be triggered. As both read requests are stream-based, they are referred only as direct and indirect.

As direct requests are issued by performing a read operation instead of a write in the respective configuration address, they must be handled differently. Thus, the *Direct Data Fetch* process is triggered. This process accesses the Buffer Table to retrieve the respective read pointer, reads the data from the FIFO buffer in Shared Memory and returns it to the CPU. Furthermore, in order to respect the request queuing order, the number of indirect requests stored in the request queues is informed to *Direct Data Fetch* process when a direct request is identified (not shown in Figure 4.11 for clarity purposes). Thus, the data returns to the CPU only after processing the informed number of indirect requests. Lastly, the *Credit Update* process is triggered to send 4-bytes credit to the sender process.

When an indirect request is identified, it is stored in queues. The *Request Selection & Processing* process selects the requests from queues using a round-robin policy. Then, the read pointer is retrieved from the respective buffer descriptor in the Buffer Table, and the *Data Fetch* process is triggered to read the data from the FIFO buffer and write it in the destination address, which can be either in Private or Shared Memory. Additionally to the read pointer, the credit threshold value is retrieved from Buffer Table. This value is set at connection set-up, when initializing the Buffer Table, and defines the number of reads to perform before sending credit to the sender process. Therefore, the *Credit Update* process is triggered when the number of reads achieve the credit threshold value or when the last data was read.

The last possibility identified by the *Request Decode* process is the initial credit generation (Table 4.4). In this case, the *Credit Update* process is triggered directly with the number of credits provided in the request.

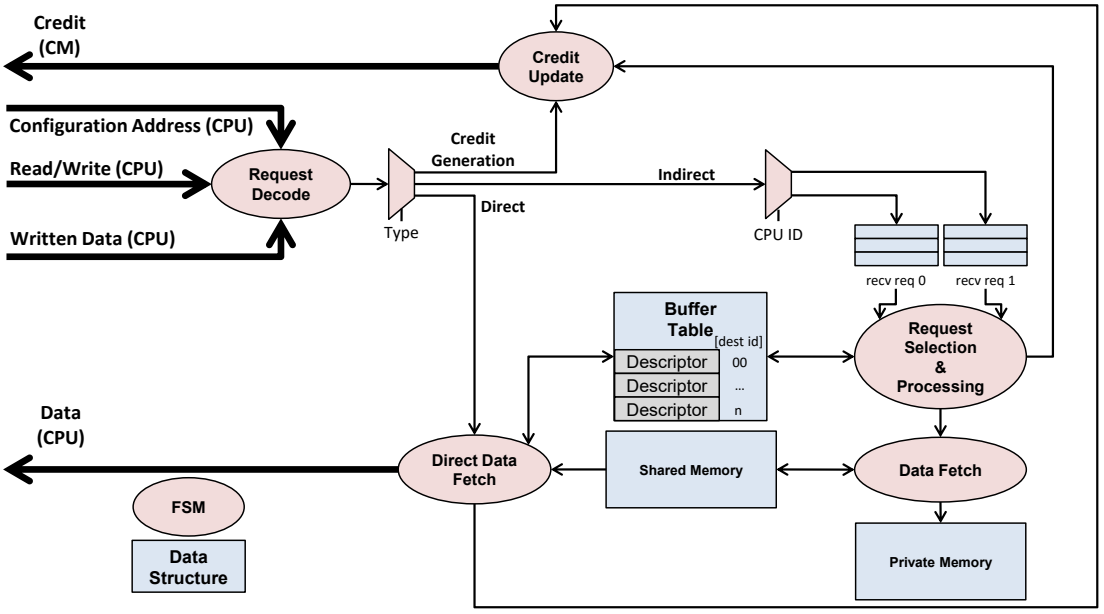


FIGURE 4.11: Buffer Manager Read functional description.

4.2.8 Credit Manager (CM)

The Credit Manager module implements the credit-based control flow used in the communication. To do so, it interacts with Network Interface (NI), Buffer Manager Interface (BMI), Buffer Manager Read (BMR) and Credit and Connection Tables, as shown in Figure 4.12. The NI is responsible for forwarding the credit packets to CM and also for sending them to the network. The BMI can read and update credits in data transfers, while the BMR only perform credit updates. The Connection Table is used to retrieve the destination port id, and the Credit Table is used to read or update credits of the local ports.

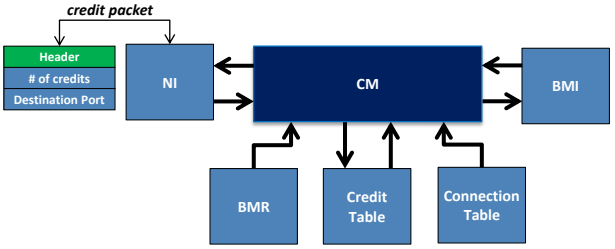


FIGURE 4.12: Credit Manager connected modules.

The Credit Manager implementation is represented in Figure 4.13. It is possible to see that data coming from NI, BMI and BMR are handled separately. The *Packet Decode* process retrieves destination port and amount of credits from the credit packet and triggers the *Credit Update* process, which performs an increment of credits for the respective port in the Credit Table.

The interface with BMI is performed by the process *BMI Request*. This process identifies the operation to be performed: read or write. When the BMI is writing credits, it means that a data transfer has been performed and the *Credit Updated* is triggered to decrease the number of available credits for the respective port. On the other hand, if the BMI is requesting credit information, the *Credit Read* process is triggered to return the number of available credits for the respective port. Thus, a single process updates the Credit Table and simultaneously manages credit increase and decrease.

On the left side, the process *BMR Request* is responsible for identifying credit updates generated by the BMR and store a credit request in a queue. The credit request contains the local port ID and the number of credits to be sent. Then, the credit requests are picked by *Credit Packing* process, which retrieves the destination ID from Connection Table, packs the data and sends the credit packet to the NI. A request queue is employed to avoid the BMR to block after generating credit updates. As the NI is a shared resource, this scenario may occur when sending the credit packet at the same time that the NI is being used by another hardware module.

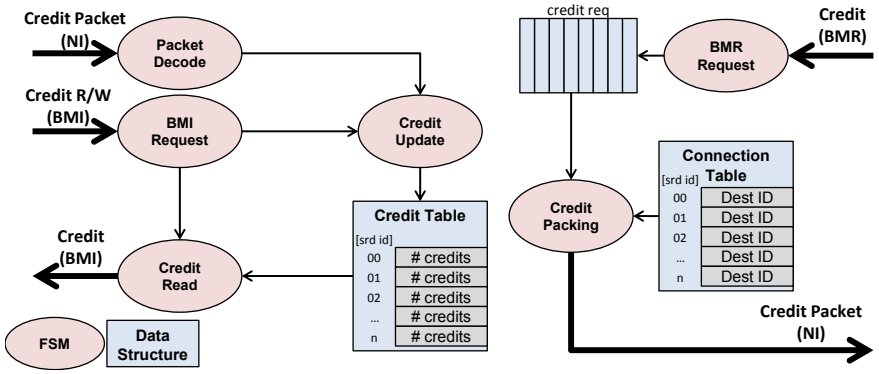


FIGURE 4.13: Credit Manager functional description.

4.3 MCAPI Modifications

The modifications in the MCAPI implementation are performed in the *opening* and *closing* steps of communication set-up phase and in the send and receive packet channel MCAPI functions. The modifications in the communication set-up phase are performed

to assure that the BMM is configured according to Section 4.2.2, while the modifications in `mcapi_trans_pktchan_send` and `mcapi_trans_pktchan_recv` functions are performed to take advantage of BMI and BMR modules. As the endpoints encodes port, CPU and Cluster IDs, they can be directly mapped to an available port ID in the BMM. Thus, when executing MCAPI functions, the decoding of an endpoint tuple to a port ID is a simple shift.

During the *opening* step, the modifications are performed in the `mcapi_wait` operation for both communication sides. In the sender side, the following lines are added:

```
unsigned int local_id = (sender_endpoint & 0xFFFF);
update_conn_tab(local_id, receiver_endpoint);
```

This function writes the `receiver_endpoint` value in the connection table. As mentioned in Section 4.2.1, the Tables are mapped in the CPU address space. Thus, to address the correct port ID, the `sender_endpoint` has the domain ID masked (16 MSBs).

The code added in the MCAPI implementation for the *opening* step in the receiver is the following:

```
unsigned int local_id = (receiver_endpoint & 0xFFFF);

update_conn_tab           (local_id, sender_endpoint);
update_buff_tab           (local_id, base_addr, size, credit_th);
initial_credit_generation (fifo_size, local_id);
```

Similarly to the sender side, the function `update_conn_tab` initializes the Connection Table accordingly. Additionally, the function `update_buff_tab` is used to initialize the Buffer Table for the respective endpoint. However, as each memory write supports only a 32-bit word, the buffer descriptor need two write accesses to be initialized. In the first access, the function writes the base address of the FIFO buffer, which is a 32-bit address in the Shared Memory. As the FIFO buffer is empty at the initialization, the read and write pointers also receive this value. The second write is responsible for initializing the FIFO buffer size and credit threshold fields. The size is coded in the 16 MSBs and the credit threshold value in the 16 LSBs. Finally, the MCAPI implementation uses a function to write the initial credit generation request for BMR, informing the `fifo_size` as the number of credits to be sent.

For the *closing* step, the `mcapi_wait` operation is modified only in the receiver side. This modification assures that the credits for the respective port in the sender side are reseted by executing the following lines:

```
unsigned int local_id = (receiver_endpoint & 0xFFFF);
initial_credit_generation ((fifo_size*-1), local_id);
```

This function is called after changing the receiver endpoint status to `CLOSED`, which guarantees that both sender and receiver performed the *closing* step.

In the data transfer phase, the modifications comprise changing the functions used to write and read data through the *FIFO API* by functions to write direct and indirect stream-based requests. Furthermore, two new events were created to take benefit from the *flag* parameter in indirect stream-based requests (Tables 4.3 and 4.4). These events are called `SENDER_PACKET` and `RECEIVER_PACKET`, and are represented by the values 9 and 10, respectively, following the codes presented in Table 3.1.

The `mcapi_trans_pktchan_recv` code becomes very short, as follows:

```
unsigned int local_port;

// Decodes the endpoint into port, node and domain IDs. Only port ID
    used in this case.
mcapi_trans_decode_endpoint(receive_handle, NULL, NULL, &local_port);

// Received size retrieved with a direct stream-based request.
*received_size = com_api_stream_based_direct_read(local_port);

// Creation of a indirect stream-based request to receive the entire
    buffer.
stream_based_indirect_read((int)sys_buffer_addr, *received_size,
    local_port, RECEIVER_PACKET);

// Uses the Event Synchronizer to wait the end of packet reception.
wait_stream_based_transfer_synch(local_port, RECEIVER_PACKET);

// Returns the System Buffer address to the application.
*buffer = (void *)sys_buffer_addr;
```

As it is showed, only three functions are used to perform the packet reception. The first one is a direct stream-based request to retrieve the packet size, as discussed in Section 2.2.4. Then, this size is used as a parameter when creating the indirect stream-based request. In this request, the System Buffer address is the destination address (`sys_buffer_addr`), since the MCAPI implementation must provide the buffer containing the data to the application. Thus, the BMR will copy the data from the respective FIFO buffer to the System Buffer. Furthermore, the `RECEIVER_PACKET` is informed as flag, and used later as the expected event for the ES in the function `wait_stream_based_transfer_synch`.

The implementation of the `mcapi_trans_pktchan_send` follows the same idea: send the packet size through direct stream-based request, send the entire packet through indirect stream-based request and inform the ES to wait for the `SENDER_PACKET` event.

However, as the requests in the BMI are selected only if there are enough credits, packets that are larger than FIFO buffers would never be processed (Section 4.2.5). Thus, a size control is implemented in this function to split the packet into several smaller size requests.

Despite only taking advantage of BMM for packet channels, the other two communication modes can also be fully handled by the BMM due to the different request types it supports. MCAPI messages specify source and target address and a transfer size, which can be handled with address-based requests. Scalar channels require dedicated connections and exchange only fixed data sizes. These characteristics are supported by direct stream based requests. Thus, the MCAPI implementation only needs to decode the channel identifier into a port ID and create the respective requests to perform send and receive operations. Therefore, as the BMM configuration can be performed with a maximum of 3 CPU write operations in memory-mapped registers, the overhead induced by the MCAPI implementation is limited and do not impact performance.

Chapter 5

Experimental Results and Validation

This chapter describes the evaluations performed to characterize the mechanisms proposed in Chapters 3 and 4, as well as the evaluation of MCAPI implementation described in Chapter 2 in terms of memory footprint. Furthermore, the performance gains obtained with the proposed mechanisms are validated through the execution of video processing and path calculation benchmarks in Section 5.5. The environment set-up is described in Section 5.1, while Sections 5.2, 5.3 and 5.4 present the characterization results for MCAPI implementation, Event Synchronizer and Buffer Manager Mechanism, respectively.

5.1 Simulation Environment

The results presented in this chapter were obtained through simulations using a SystemC [66] model of the reference architecture (Section 1.4), which was developed during this Thesis. The model is described at TLM (Transaction-Level Modeling) level with timing annotation, with the CPU core wrapping a MIPS R3000 ISS model [67]. The modules are connected through socket ports, exchanging generic TLM transactions [66]. At network level, each flit is represented by one TLM transaction. However, for NoC flits, the transaction is a specific class that models the different flit types. Inside Clusters, the transactions are managed mainly by a generic bus, which forwards the transactions according to their destination addresses and the cluster address map.

Table 5.1 presents the parameters used in the simulations. The number of clusters differs according to the scenario being evaluated. On the other hand, the other parameters are fixed. Indeed, the Private Memory size can be set to values up to 128 kilobytes (kB) before running the simulation, while the Shared Memory size supports values up to 1024 kB. By default, the simulations are performed using the maximum sizes. However, if a given application requires larger memory sizes, e.g., video processing, the CPU address map can be modified to fit with the new sizes.

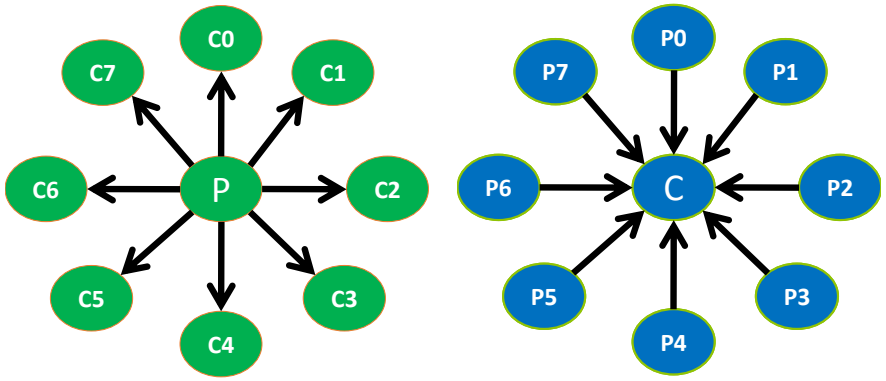
TABLE 5.1: Architecture parameters used in the simulations.

Parameter	Value
CPU Frequency	200 MHz
NoC Frequency	500 MHz
Number of Clusters	2 to 16
Number of CPUs per Cluster	2
Private Memory Maximum Size	128 kB
Shared Memory Maximum Size	1024 kB

5.1.1 Simulation Scenarios

The simulations are performed with two objectives: characterize the performance of the proposed mechanisms and evaluate the performance gains at application level. The mechanisms characterization is performed with the "*ping-pong*" benchmark, which is usually employed to characterize latency and throughput in works such as [12, 68, 69]. This application consists of sending a message from one process to another and wait for the reply. In this context, the process that generates the first message is called *ping*, while the process that receives and replies the message is called *pong*. Furthermore, scenarios with multiple *pong* processes are evaluated, as depicted in Figure 5.1. These evaluations are performed to represent a single-producer multiple-consumer scenario (Figure 5.1(a)) when the *ping* process is sending the first message, and a multiple-producer single-consumer scenario (Figure 5.1(b)) when the *pong* processes are sending their replies.

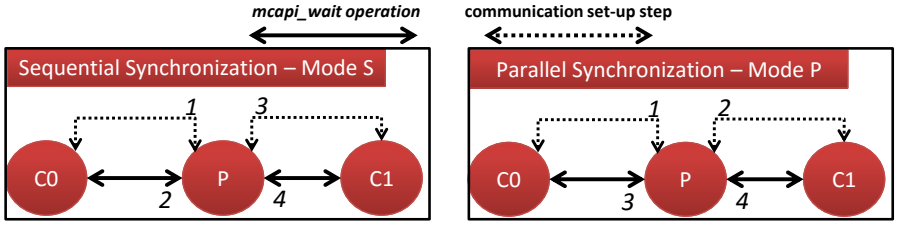
In addition to the multiple number of connections (*pong* processes), two synchronization schemes were evaluated in the Event Synchronizer characterization. These schemes are depicted in Figure 5.2. In the sequential synchronization scheme (Mode S), all the communication set-up steps (*connect*, *opening* and *closing*) and their respective `mcapi_wait` operations are performed with each pong process before communicating to the next one (Figure 5.2(a)). On the other hand, in the parallel synchronization scheme (Mode P), each step is performed with every pong processes (1 and 2 in Figure 5.2(b)) before performing the respective `mcapi_wait` operation (3 and 4 in Figure 5.2(b)) with



(a) (b)

FIGURE 5.1: Ping-Pong application with multiple connections. (a) The "ping" process sends data to multiple "pong" processes. (b) The "pong" processes send data to the "ping" process.

the respective process. These schemes reflect how the programmer might code the application/benchmark, i.e., the non-blocking operation does not need to be performed right after the completion of a non-blocking function.



(a) (b)

FIGURE 5.2: Sequential and parallel synchronization schemes for the ping-pong benchmark.

Finally, the applications *susan* and *dijkstra* were used to validate the performance gains obtained with the proposed mechanisms. These applications are part of MiBench suite [70] and were modified by [71] to employ MCAPI packet channels in inter-process communication. *Susan* is an image recognition package used for recognizing corners and edges in magnetic resonance images. This type of image processing is common in real world applications and could be employed for a vision based quality assurance application. This benchmark provides image adjustments for threshold, brightness, and spatial control. The *dijkstra* benchmark is used to calculate the shortest path between every pair of nodes in a given graph using repeated applications of the Dijkstra's algorithm [72].

5.2 MCAPI Memory Footprint

Implementing MCAPI with low memory footprint is mandatory, since this characteristic is one of the main goals of MCAPI. However, it is not only the code size that occupies memory space in the MCAPI implementation. The structures also contribute for a significant part of the total memory footprint. Thus, both aspects are taken into account in the evaluations and are detailed in the following sections.

5.2.1 Transport Layer Code

The evaluation of code size and comparison with the other APIs from the reference architecture are presented in Table 5.2 and Figure 5.3, respectively. The MCAPI implementation has two layers: *mcapi* and *mcapi_trans*. The first layer provides the MCAPI functions calls as defined in the specification, e.g `mcapi_pktchan_send`. Additionally, this layer performs tests in order to cover the error conditions determined by the specification. Then, after executing the error checking functions, the *mcapi* layer calls a *mcapi_trans* function, which implements the respective functionality.

TABLE 5.2: Code sizes for different implementations of software API layers in the reference architecture.

API	BMM and ES		DMA and ES		DMA		Pure Software	
	Size (bytes)	# instructions	Size (bytes)	# instructions	Size (bytes)	# instructions	Size (bytes)	# instructions
boot	176	44	176	44	176	44	176	44
mips_debug	1316	329	1316	329	1316	329	1316	329
libc	504	126	504	126	504	126	504	126
mips_com_api	8124	2031	8124	2031	8124	2031	8124	2031
fifo	1480	370	1480	370	1480	370	1480	370
mcapi	4428	1107	4428	1107	4428	1107	4428	1107
mcapi_trans	**13460**	3365	**13320**	3330	**12520**	3130	**12432**	3108
Total	**29488**	7362	**29348**	7337	**28548**	7137	**28460**	7115

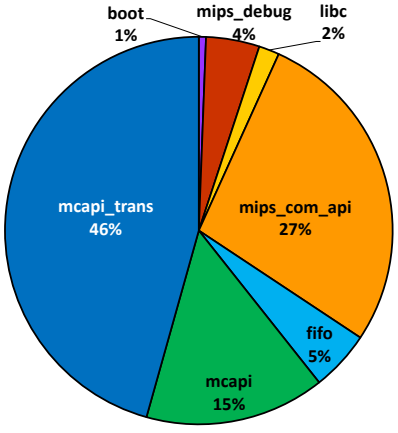


FIGURE 5.3: Software layers proportional contribution in the entire software stack.

The error checking process introduces a processing overhead that is intrinsic to the MCAPI implementation, since these errors are covered by the specification. An example would be at endpoint creation, when the MCAPI implementation must check for a valid port ID (e.g. port ID <= MCAPI MAX ENDPOINT). Table 5.3 shows the number of cycles and instructions used to execute the `mcapi_pktchan_send_open_i` function. The number of cycles used to execute the functional implementation corresponds for 54% of the total function execution time and the number of instructions corresponds for 41% of the instructions, i.e., 46% and 59% of execution time and memory footprint overheads, respectively. These overheads are not negligible for functions that execute *opening* and *closing* set-up steps, since they are used basically for changing endpoint status and filling up a MCAPI request structure. A solution to eliminate these overheads would be assuring that all the parameters are correct prior to the executing these steps, and then, implement these functions without error checking.

TABLE 5.3: Number of cycles spent by each function in the transport layer to complete a `mcapi_pktchan_send_open_i` function.

Function Purpose	**Function Call**	**# Cycles**	**% of Total Cycles**	**# of instructions**	**% of Total instructions**
Functional Implementation	trans_send_open	703	51%	123	35%
	Get domain and node ids	41	3%	22	6%
Error Checking	Check initialized	43	3%	37	10%
	Check valid endpoint	156	11%	46	13%
	Check channel type	105	8%	35	10%
	Check send endpoint	111	8%	32	9%
	Check open pending	104	8%	30	8%
	Check channel open	107	8%	30	8%
	Total	1370	100%	355	100%

In terms of code size, the MCAPI implementation code size ranges from 16.5 kB to 17.5 kB (considering both layers), which represents 59.2% and 60.7% of total code size for the entire software stack. Matilainen et. al [9] reported a total code size of 25 kB, using 1450 lines of C code for the transport layer (equivalent to *mcapi_trans*), while the Authors in [5] used around 3700 lines of code to implement the transport layer. For comparison sake, the MCAPI implementation performed by this work uses around 2500 lines of C code (without comments). Although only the packet channel functions are implemented in the *mcapi_trans* layer, most functions used for error checking and endpoint management can be reused to implement the other communication modes. Therefore, implementing messages and scalar channel functions will not increase the code size significantly. Thus, the MCAPI implementation present similar characteristics to the state-of-the-art.

The variation highlighted in Table 5.2 for *mcapi_trans* is related to the code modifications described in Sections 3.3 and 4.3. The functions that contribute the most

for this variation are presented in Table 5.4, as well as their code sizes for different implementation versions.

The version that uses less memory is the pure software implementation. In this version, the differences in code size are not significant when compared to the DMA version, with only two functions presenting lower memory occupation: `pktchan_send` (line 3) and `init_paths` (line 6). The difference in the `pktchan_send` function is explained by the size control when sending blocks of data. This modification was performed in order to send blocks of data according to the available space in the FIFO. As the pure software implementation uses only the `fifo_write` function (sends only one 32-bits data), the code is a simple `for` loop of the packet size. The `init_paths` has lower size for pure software because there is no need to initialize the DMA routing table. Nevertheless, the code size is increased in 88 bytes (0.3% of total size) in the MCAPI implementation and in 748 bytes (2.7% of total size) in `fifo` layer for the addition of `fifo_write_block` function.

The performance gains were evaluated through the *"ping-ping"* application with 1 pong process and is presented in Figure 5.4. The metric used is the total execution time for different packet sizes. It is possible to see that, using the DMA, the performance is increased for packet sizes larger than 16 bytes (4 words) and can achieve up to 23% for packet sizes of 1 kB or higher. However, for smaller sizes, the processing overhead decreases the overall performance. Thus, the DMA is used only when sending packets of 4 or more 32-bits words.

TABLE 5.4: Memory footprint for the different versions of functions affected by the use of BMM, DMA and ES.

	Function	BMM and ES	DMA and ES	DMA	Pure Software
		Size (Bytes)	Size (Bytes)	Size (Bytes)	Size (Bytes)
1	mcapi_trans_pktchan_connect_i	992	992	952	952
2	mcapi_trans_pktchan_recv	152	248	248	248
3	mcapi_trans_pktchan_send	280	244	244	188
4	mcapi_trans_test_i	2280	2168	1760	1760
5	mcapi_trans_wait	1996	1884	1524	1524
6	mcapi_trans_init_paths	56	56	56	16

With the modifications performed to take advantage of the Event Synchronizer the `mcapi_wait` (line 5) and `mcapi_test` (line 4) operations had their code size increased. Both functions represent an increase of 768 bytes (2.7% of total size) when compared to the DMA column. This increase is a consequence of the modifications described in Section 3.3. Finally, the modifications described in Section 4.3 also affected `pktchan_send` and `pktchan_recv` function sizes. However, as the `pktchan_recv` function was simplified, the code size was decreased in 96 bytes. Overall, the code size was increased in 940 bytes (3.3% of total size) when compared to the DMA version. Nonetheless, the instructions count increase does not imply higher execution times, as showed in Sections 5.3

and 5.4. Indeed, the additional code is required to configure the proposed mechanisms accordingly, which offloads the inter-process communication from software and, hence, leads to a lower number of executed instructions.

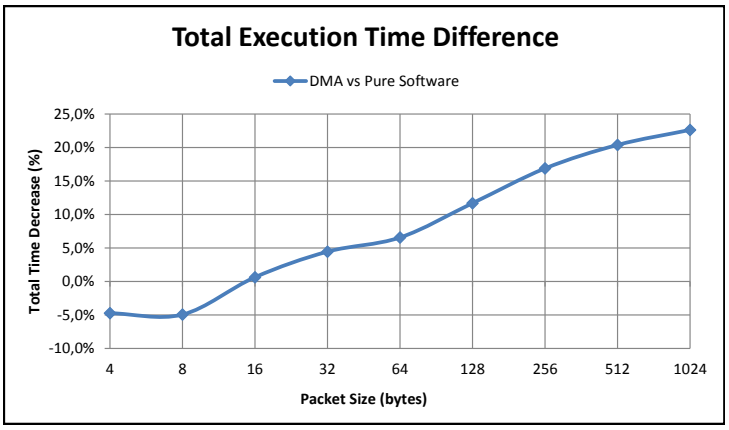


FIGURE 5.4: Difference in the total execution time for the ping-pong application when using DMA or pure software for data transfers.

5.2.2 MCAPI Structures

The memory footprint for the MCAPI structures are related to: number of nodes, number of FIFOs per domain, maximum FIFO size, number of endpoints, maximum number of requests per endpoint and System Buffer size. The values of these parameters can be modified in the MCAPI implementation header file. However, the size of domain, node and endpoint attributes as well as the request structure size are fixed. These values are detailed in Table 5.5.

TABLE 5.5: Size of MCAPI structures placed in the Shared Memory.

Domain Structure	Size (Bytes)
domain entry	20
node entry	20
endpoint entry	8
endpoint attributes	64
request structure	32

On the other hand, the total size of the other structures depend on the application constraints. Considering the case where the highest amount of memory was used, all the structures occupied around 37 kB of the Shared Memory (Table 5.6), using FIFOs of 128 Bytes. However, this size can be dramatically decreased by tuning the amount of resources according to the application (e.g. reducing the number of FIFOs, System Buffer size, number of endpoints per node, etc).

TABLE 5.6: Evaluation of data structures memory footprint.

Parameter	#
Number of Nodes per Domain	2
Number of Endpoints per Node	64
Number of Requests per Endpoint	4
Number of FIFOs	64
System Buffer	1
Data Structure	**Size (Bytes)**
Attributes Structure	9276
Request Structure	16384
FIFO Structure	8192 (128B per FIFO)
System Buffer	4096
Total	37948

5.3 Communication Set-up Characterization

The Event Synchronizer aims to decrease the processing and traffic overheads due to polling operations used by the software implementation. Thus, two metrics are used to characterize the performance gains obtained with the ES: network load and CPU load. The network load is evaluated by tracking the amount of data sent by each process through the Network Interface. In turn, total execution time and active/idle processor time are taken into account to evaluate the CPU load. Both metrics can be related to the overall system power efficiency.

The data injected in the network reflects directly the power consumption. Reducing the number of flits sent by each process decreases the power consumption in two ways: (i) less information being routed; (ii) lower bandwidth used, which avoids collision and makes the packets to arrive faster at their destination. Indeed, lower network activity decreases switching activity and consequently, the dynamic power consumption.

The idle time directly reflects in power consumption since it decreases the number of executed instructions and memory accesses, decreasing switching activity and, consequently, dynamic power consumption. Furthermore, the idle mode can take advantage from a CPU low-power state, e.g. using DVFS, if it is supported by hardware architecture.

The first evaluation has the objective to compare the results presented in Figures 2.6 and 2.8 and the results obtained with the MCAPI implementation taking advantage of the Event Synchronizer. This comparison is performed using the sequential synchronization scheme (Mode S) for the "*ping-pong*" application with one pong process. The results are showed in Figures 5.5 and 5.6.

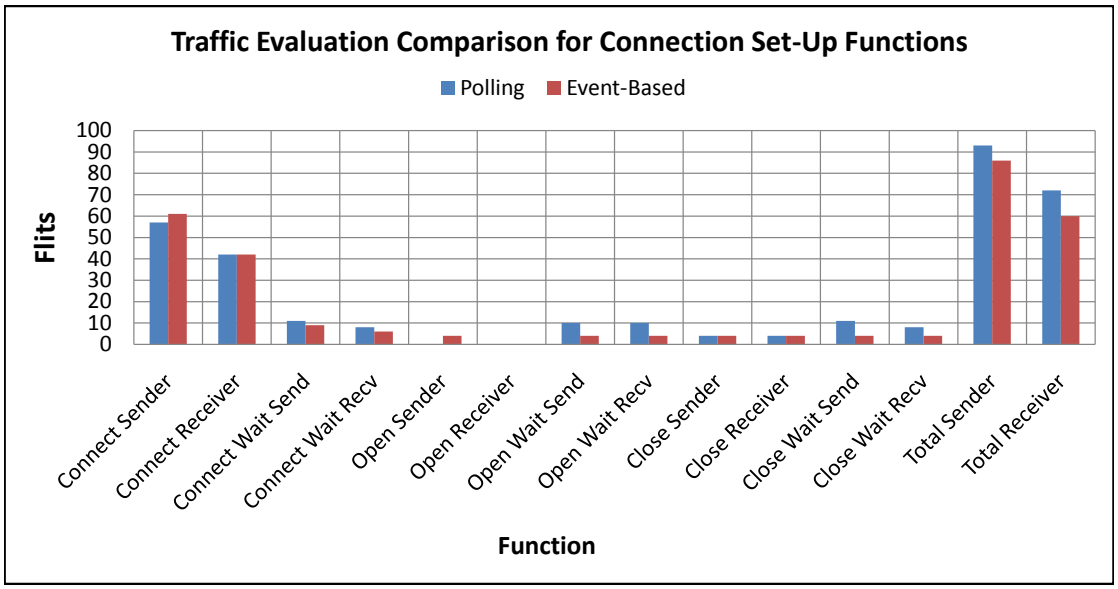


FIGURE 5.5: Number of flits sent by each connection set-up function using polling and event-based approaches.

Figure 5.5 demonstrates that the total number of flits was decreased for both sender and receiver processes, despite the slight increase in the number of flits due to the synchronization packet sent by the `connect` and `open` functions in the sender. Nevertheless, the number of flits was decreased from 93 to 86 in the sender side, while in the receiver side the decrease was from 72 to 60 flits, which represents a reduction of 7.5% and 16.7% for sender and receiver, respectively, and an overall reduction of 11.5%. This difference becomes more significant if the desynchronization is considered, as showed in Figure 5.6.

The desynchronization occurs when the communication sides do not perform a synchronization step at the same time. Thus, a desynchronization rate can be defined as the amount of time a communication process takes to start a synchronization step, relatively to the amount of processing already performed by the opposite communication process. It means that, if a receiver process starts the execution of a synchronization step only after the sender process has already performed half of the same step, the desynchronization rate would be of 50%. Similarly, if a sender process starts the execution of a synchronization step only after the receiver has already finished the execution of its respective function, the desynchronization rate would be of 100%.

To evaluate this effect, all flits sent by sender and receiver processes are considered (initialization, synchronization, etc). However, in order to decrease the influence of traffics generated by other sources, the communication set-up is performed 10 times between the ping and pong processes. Furthermore, no data is transferred, which means that only the 3 communication set-up steps are performed. The results show that, with

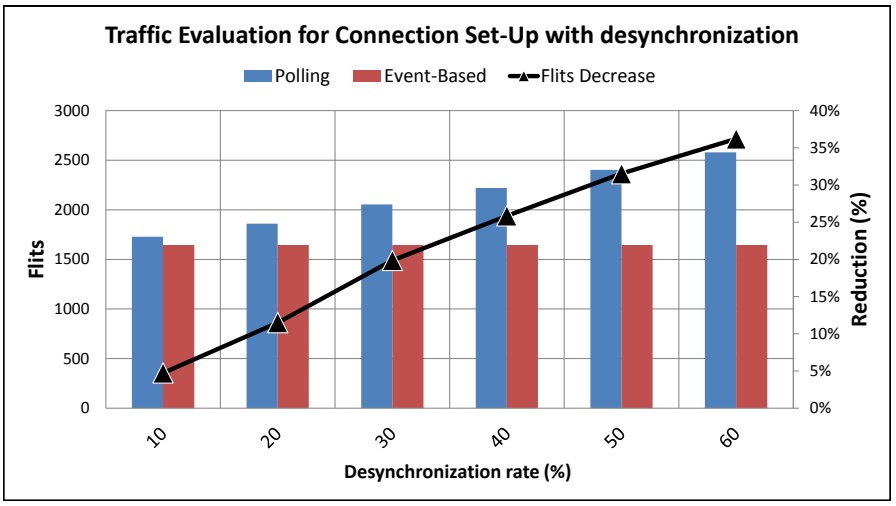


FIGURE 5.6: Number of flits sent in the communication set-up phase using polling and event-based approaches for several desynchronization rates.

the Event Synchronizer, the number of flits sent is the same regardless the desynchronization rate. Thus, when compared to the software implementation, the gain in the number of flits is linear and achieves 36.2% for a desynchronization rate of 60%.

These evaluations demonstrate that the gains obtained with the Event Synchronizer can be very significant. Therefore, several scenarios were simulated to further evaluate network and processing loads.

5.3.1 Network Load

Figures 5.7 and 5.8 present the results obtained in terms of network traffic for several numbers of pong processes and for both synchronization schemes. The number of flits sent by the ping process for different number of connections (pong processes) is presented in Figure 5.7. Although no data is transferred, up to 16 endpoints are created in the ping process, since it is supposed to send and receive data. The values of the Y-axis are showed in thousands and represent the traffic generated for the communication set-up between the endpoints used to send and receive data. The communication set-up phase was performed 64 times.

The chart shows that, compared to polling scheme, the number of flits significantly decreases in Mode P and slightly decreases in Mode S when using the Event Synchronizer. This can be explained by the fact that while the ping process performs the *connection* step with all pong processes before going to the next step (*opening*) (Mode P), the pong processes that completed the *connection* step start to perform polling operations to read the ping process status for the *opening* step, generating network traffic. On the

other hand, in the sequential synchronization scheme (Mode S), the pong processes stay blocked in the *connection* step checking if the local FIFO is initialized by performing local polling. Therefore, the decreasing in network traffic for Mode S is less significant than for Mode P due to receivers performing local polling (translated into memory accesses) instead of remote polling (translated into network traffic).

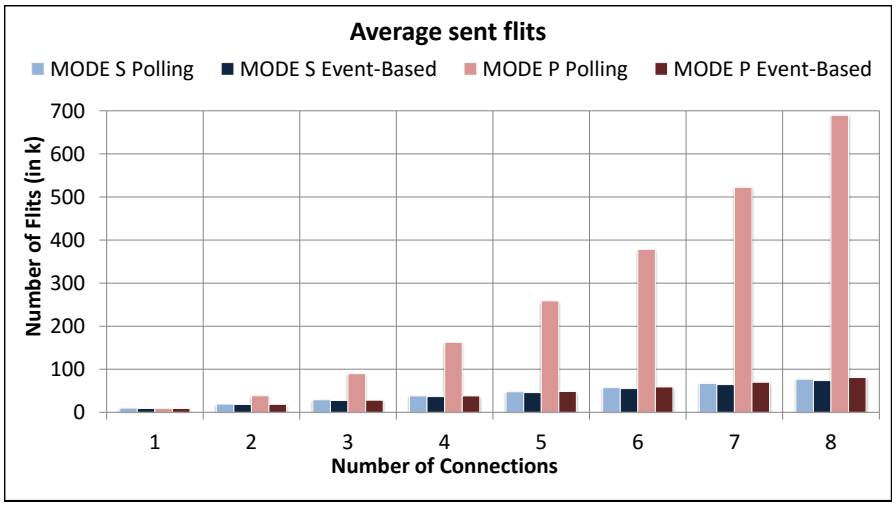


FIGURE 5.7: Number of flits sent by the ping process for different number of connections in both synchronization schemes.

Figure 5.8 shows the decrease in the flits sent by the ping process in percentage. It can be seen that with the event-based scheme (using ES) the number of flits is kept almost the same for both synchronization schemes. The reduction in the number of flits sent ranges from 4% to 7% for Mode S and from 1% to 88% for Mode P. Additionally, the number of memory access in Mode S was also decreased from 1% to 87%. Although the curves show the data only for the ping process, the pong processes also present the same gains for both synchronization schemes according to the total number of connections.

Moreover, it is important to highlight that the way the application is coded may impact the overall performance. Considering this scenario, even the application presenting the same behavior for both synchronization schemes, the parallel synchronization presents a significant higher amount of flits sent by each process. However, from the software programmer point of view, this difference may be not clear when coding the application. Therefore, the hardware optimizations co-designed with software API can compensate this difference without impacting coding complexity.

5.3.2 CPU Load

When using pure software implementation, the polling operations in the synchronization steps are performed by the processor. It has been stated in Chapter 3 that this approach

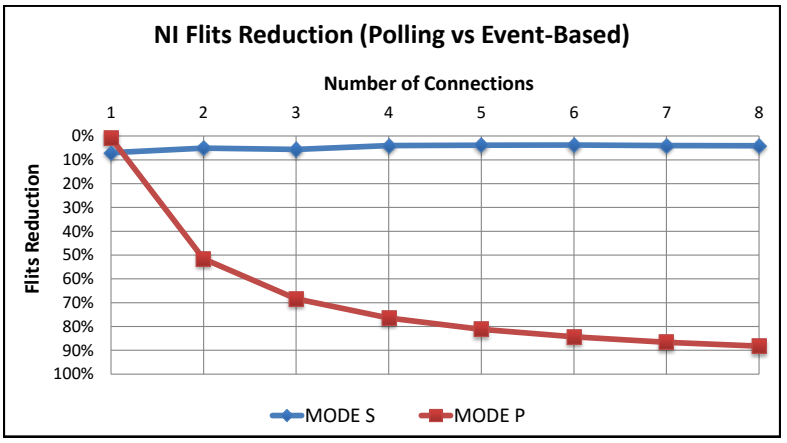


FIGURE 5.8: Decrease in the number of sent flits when using ES in both synchronization modes.

is not optimal and leads to resources misuse and, consequently, to low power efficiency. On the other hand, the event-based approach has higher power efficient, since it allows the processor to execute other tasks while waiting for an event or to switch to a low-power/idle state. Thus, the total execution time and idle/active time ratio was used to evaluate the CPU processing load.

Figure 5.9 presents the execution time for both synchronization schemes according to the number of connections when using polling and event-based approaches. The total execution time is decreased by 1% in average when using the Event Synchronizer for both synchronization schemes. The reason for not achieving higher gains is explained by the fact that the communication set-up steps cannot be speed-up, since the ES is not able to generate the condition to complete a step faster. In other words, the processors will execute the same instructions to perform the connection set-up in both communication sides either using or not the ES. As an example, considering that the ping process executes the `pktchan_connect_i` function at *t0*, the `mcapi_wait` operation has to wait until the endpoint in the pong process change its status to *connected*, which will happen at *t1* and will not be influenced by the Event Synchronizer. The same idea can be applied to the other polling processes.

However, by taking advantage of the event-based approach, the CPU can program the ES to be notified when the respective event is received and, in the meantime, perform other actions (e.g., task preemption in case of multi-thread operating system or decrease CPU voltage and frequency to save power). In the scope of this Thesis, the reference architecture model CPU has the clock signal stopped, which is called “idle” state. Figure 5.10 presents the evaluation in terms of idle time for the previously mentioned scenario.

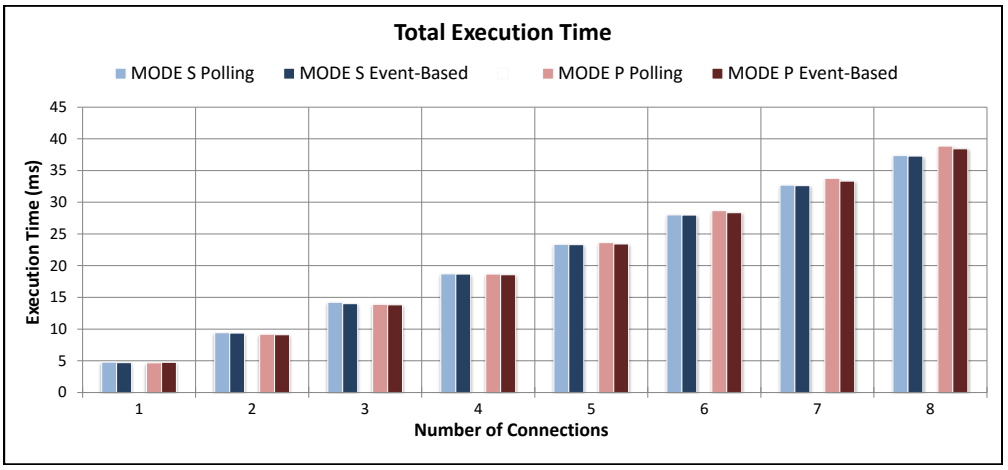


FIGURE 5.9: Total execution time of ping-pong application for a different number of connections in both synchronization modes.

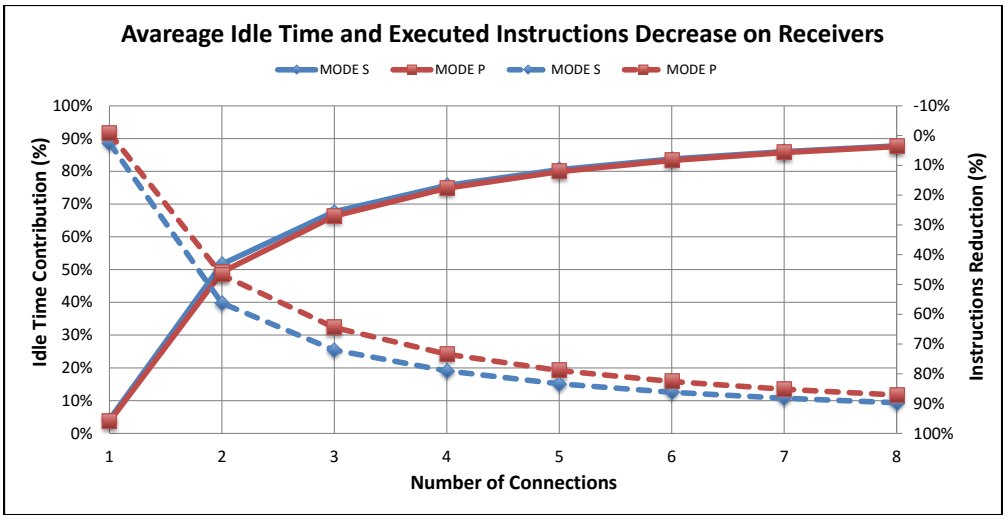


FIGURE 5.10: Relation between idle time (solid line) and reduction in the number of executed instructions (dotted line) for both synchronization modes.

In addition to the idle time represented by the left Y-axis and solid lines, the decrease in the number of executed instructions is represented by the right Y-axis and dashed lines. The number of executed instructions reflects the CPU switching activity and memory accesses. As expected, the decrease in the number of executed instructions is directly proportional to the idle time contribution over the total execution time. Figure 5.10 shows that the idle time increases significantly with the number of connections for both Modes. The curves show that with 2 connections the idle time already contributes for around 50% of the total time execution. With 8 connections, the idle time achieves around 88% of the total execution time. A synthesis of idle, active and total execution times is presented in Figure 5.11. Therefore, the CPU load can be dramatically decreased by taking benefit of the Event Synchronizer, which gives opportunity to increase power

efficiency.

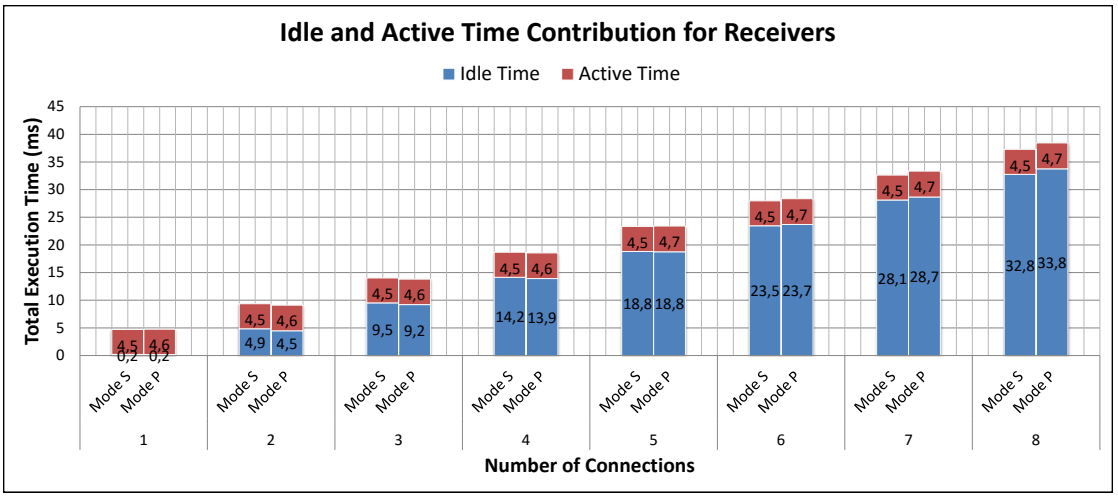


FIGURE 5.11: Total execution time divided between idle and active times for different number of connections in both synchronization modes.

5.4 Data Transfer Characterization

The Buffer Manager Mechanism targets to decrease processing and traffic overheads by implementing FIFO management and a credit-based flow control policy in hardware. This mechanism is used in the data transfer phase. Therefore, two basic metrics used to evaluate communication performance are employed: throughput and latency. Moreover, the performance gains obtained with the BMM are evaluated by analyzing network traffic, communication time and total execution time.

Throughput and latency measurements are performed with one connection in the "*ping-pong*" application. However, for throughput evaluations, only the "ping" process sends data, since the throughput measurement can be performed with a single communication channel. On the other hand, for network traffic and communication/total time measurements, both processes send data to each other for a number of ping-pong connections varying from 1 to 8. Additionally, since the ES is used in all the evaluations and both sequential and parallel synchronization schemes present similar performance with ES, only the sequential synchronization scheme is considered.

5.4.1 Throughput Evaluation

The initial throughput evaluation is performed by transferring 32 kB of data from ping to pong process for different packet sizes. Figure 5.12 presents the throughput obtained

when using Pure Software implementation, DMA and BMM. Furthermore, the curves are compared to the theoretical throughput limit, represented by the dotted line (TP Limit). This limit is calculated by taking into account the CPU frequency (200 MHz) and the fact that it takes 2 clock cycles to write each data (32 bits) in the bus. Therefore, the maximum achievable throughput is 3200 Mbps. The evaluations are performed with a FIFO size of 32 bytes.

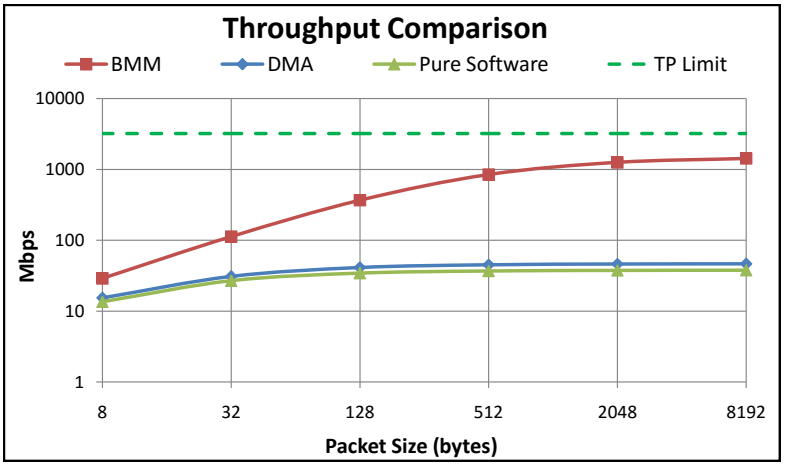
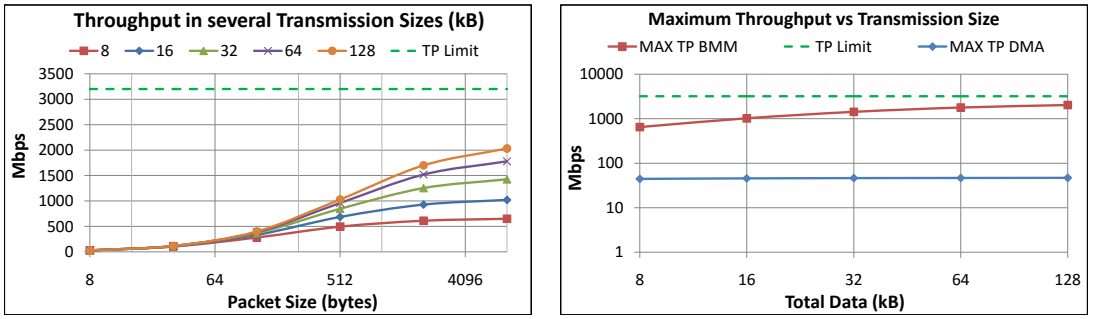


FIGURE 5.12: Throughput comparison between BMM and DMA for 32 kB of transmitted data at different packet sizes.

The Y-axis is represented in logarithmic base due to the difference between the obtained throughputs. It is possible to see that the throughput increases for larger packet sizes, since the application divides the total amount of data in fewer requests and, consequently, generates less processing overhead and finishes the data transfer faster. With pure software and DMA implementations, the FIFO available space is used by the API level to create a packet (software implementation) or request (DMA) of its respective size. On the other hand, when using the BMM, the FIFO size (fixed parameter) determines the request size. Thus, it can be assumed that larger FIFO sizes can increase the throughput, specially for smaller packet sizes. However, since indefinitely increasing FIFO size is not realistic, the throughput for large packets (e.g., 2kB) do not change significantly, maintaining the same curve behaviors.

The throughput achieved with DMA and FIFO API ranges from 15 Mbps to 46 Mbps, while the throughput achieved with pure software implementation ranges from 13 Mbps to 38 Mbps. This difference shows that, even with a DMA being in charge of data transfer, the throughput is limited by the FIFO control software implementation. On the other hand, when using the BMM, the achieved throughput ranges from 29 Mbps to 1425 Mbps, i.e., a throughput up to 30 times higher. Moreover, since the performance with a DMA is higher than pure software implementation, the next comparisons are performed only between DMA and BMM.

To further characterize BMM throughput, scenarios with different amounts of transferred data are evaluated. Figure 5.13 presents the results for data transfers of 8, 16, 32, 64 and 128 kB. In Figure 5.13(a) the curves show that the maximum throughput achieved with BMM gets closer to the theoretical limit as the amount of transferred data is increased. Figure 5.13(b) compares the throughput achieved with BMM and DMA for different transfer sizes with a packet size of 8 kB. While the BMM throughput increases for higher amounts of transferred data, the throughput using the DMA stays constant, which exposes the overhead imposed by the FIFO API.



(a) (b)

FIGURE 5.13: Maximum achievable throughput for different amounts of transmitted data.

Finally, the throughput efficiency is evaluated in Figure 5.14. The efficiency is calculated as the ratio between the number of useful data bytes and the total number of bytes sent through the NoC, which includes the protocol overhead. The evaluation was performed for 32 kB of useful data transmitted. The total number of bytes takes into account only the ping process and considers the channel set-up messages. The result shows that the BMM doubles the throughput efficiency, since the overhead induced by the software implementation due to the FIFO pointer exchanging is removed. The remaining overhead is produced by the header of each packet containing data, the packet informing the transfer size and the synchronization packet. Thus, a lower overhead impact is expected for larger packets, since the number of transfer size and synchronization packets is decreased.

5.4.2 Latency Evaluation

The latency evaluation is performed by measuring the round-trip time (RTT) in two scenarios: (i) a single packet of variable size and (ii) transmission of 8KB of data with different packet sizes. The results are presented in Figure 5.15.

Figure 5.15(a) presents the round-trip time comparison between DMA and BMM. The RTT is measured from the `mcapi_pktchan_send` function call to the completion of

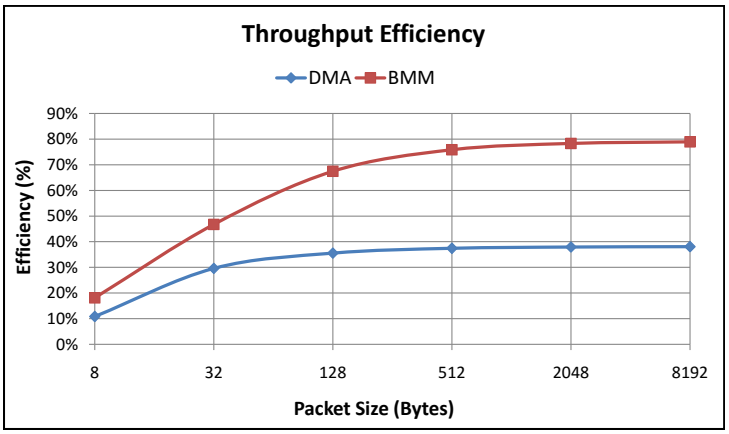
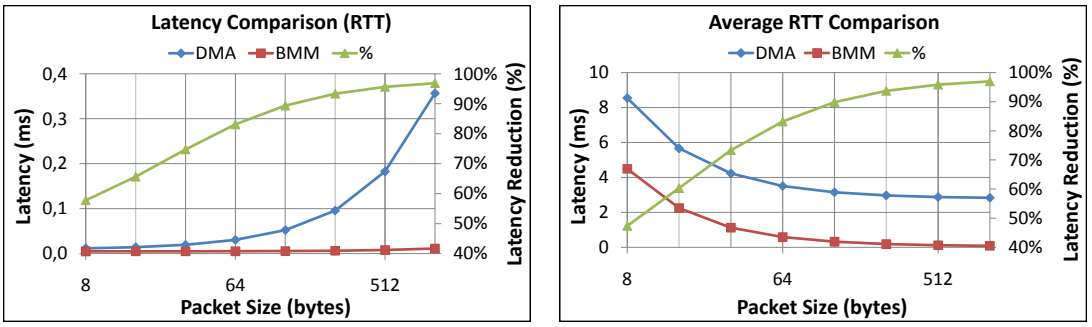


FIGURE 5.14: Throughput efficiency comparison between BMM and DMA.



(a) (b)

FIGURE 5.15: Round-trip time evaluation. (a) Round-trip time for 1 packet of different sizes. (b) Round-trip time for 8 kB of data transmitted at different packet sizes.

the `mcapi_pktchan_recv` function in the ping process, thus taking into account receive and send functions performed by the pong process. The time spent to complete one "*ping-pong*" exchange by the implementation using the DMA ranges from 11.4 ms (8 bytes packet) to 356.8 ms (1 kB packet). On the other hand, the implementation taking benefit from BMM presented RTT times ranging from 4.8 ms to 11.1 ms. This difference represents a decrease around 60% for smaller packets and up to 97% for larger packets. It shows that the BMM is able to completely overcome the overhead introduced by the software implementation. Furthermore, this result shows that the BMM can dramatically increase the performance of applications that use scalar channels, since the streamed data will be delivered a lot quicker to the receiver endpoint.

Similar gains were obtained when evaluating the round-trip time for a transmission of 8 kB of data with different packet sizes, as shown in Figure 5.15(b). In this scenario, all the data is considered a "single packet", i.e the pong process replies to the ping process only after receiving all the 8 kB of data. This evaluation mimics the behavior of applications that needs to exchange large amounts of data at once, such as video

processing. The total latency is higher for smaller packet due to the higher number of packets needed to complete the entire transmission. In summary, the latency is decreased up to 97% and 80% in average.

5.4.3 Network Load

The network traffic is evaluated by measuring the amount of data exchanged through the NoC. The evaluation is performed with the "*ping-pong*" application exchanging 128 kB of data with each ping-pong connection. Two parameters are considered: packet size (from 32 to 1024 bytes) and number of connections (from 1 to 8). However, for clarity purposes, Figure 5.16 present the number of flits sent by the ping process for three different number of pong processes. Nevertheless, the average gain curve is calculated considering all scenarios.

The number of connections is denoted by the suffix in DMA and BMM series. The "2c" suffix denotes the scenario with 2 pong processes, while suffixes "4c" and "8c" are used to identify the scenarios with 4 and 8 pong processes, respectively.

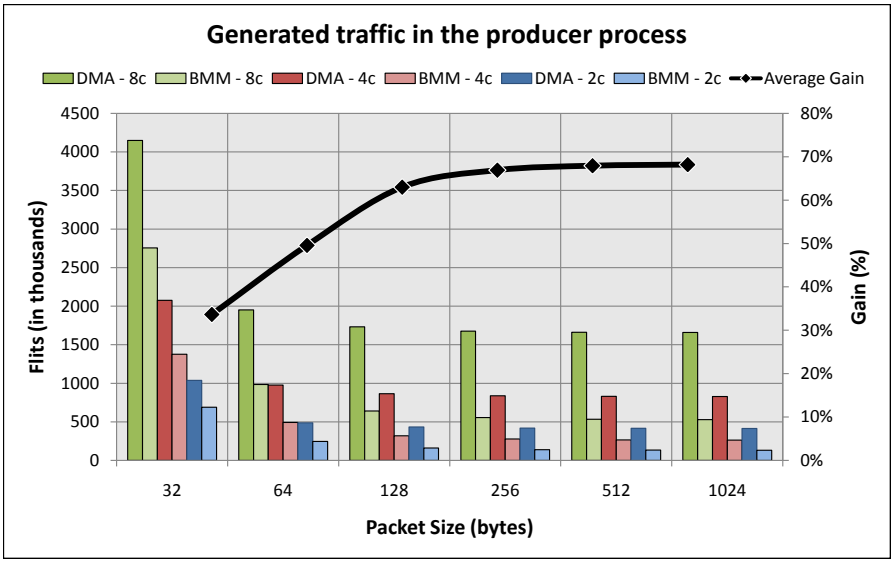


FIGURE 5.16: Total number of flits sent by for the ping process to transfer 128 kB of data.

The decrease in the number of flits sent by the ping process, in percentage, is the same despite the number of connections. Indeed, the number of flits sent by the ping process is around the sum of flits sent by all pong processes, since both sides perform the same actions. The results demonstrate that the BMM provides an important decrease in the total number of flits, from 33.6% to 68.2% and 58.2% in average. It is also possible

to note that, around 128 bytes packets, increasing packet size has no significant impact in the total number of flits exchanged.

Furthermore, the results presented in Figure 5.16 can be compared to the throughput efficiency, shown in Figure 5.14. Considering the scenario with 8 connections and packet size of 1 kB using the BMM, the total number of flits sent by the ping process is of 528,000, which represents 66,000 flits per connection. As the useful and minimal amount of flits is 32,000 (each flit has 4 bytes), the obtained efficiency is 48.4%.

However, Figure 5.14 suggests that, for this packet size, the efficiency should be higher. This can be explained by the fact that only the ping process sends data in the throughput evaluation. On the other hand, in this scenario, the ping process sends and receive data. Therefore, in addition to the aforementioned overhead, the credit packets sent to the pong processes also contributes to decrease the global efficiency.

5.4.4 Communication and Total Execution Times

The last evaluations use communication and total execution time as metrics. These evaluations allow to characterize the impact of the BMM in the application performance. In addition to the number of ping-pong connections and packet size, the MCAPI FIFO size is also used as parameter. Three FIFO sizes are analyzed: 64, 128 and 256 bytes. Figure 5.17 compares the communication time when using DMA and BMM for different packet and FIFO sizes and for 2, 4 and 8 connections. The X-axis variables are represented in the following order, from top to bottom of the figure: packet size (from 32 to 1024 bytes, in vertical), FIFO size and number of pong processes. The amount of data transmitted between each ping-pong connection was 32 kB for all scenarios.

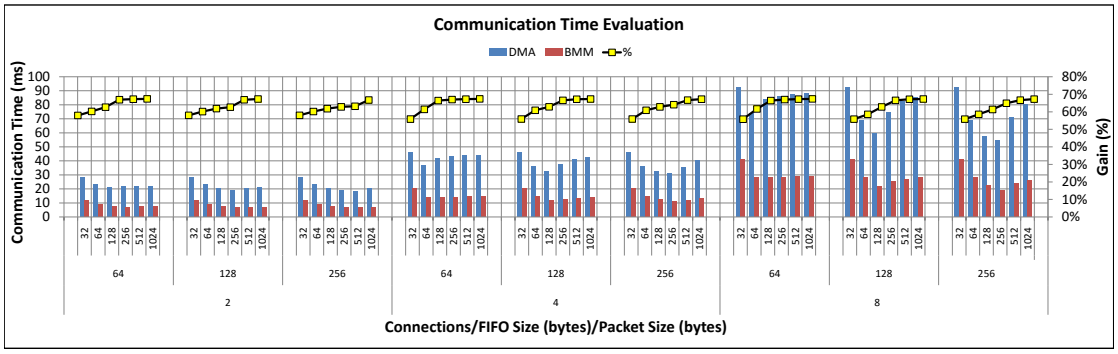


FIGURE 5.17: Communication time using DMA and BMM for different packet and FIFO sizes with different number of connections.

The curves show that the communication time using the BMM was decreased between 55% and 67%. Additionally, it can be seen that the FIFO size does not impact the

communication time for 2 and 4 connections. On the other hand, scenarios with 8 pong processes present a slight difference in the communication time for packet sizes between 128 and 1024 bytes depending on the FIFO size. It is possible to see that, for 8 pong processes, the lowest communication time is obtained when the packet size matches the FIFO size. This occurs due to the ping process writing the entire packet in the pong process FIFO and starting to send the data to the next pong process. However, when the packet size is larger than the FIFO size (i.e., the API has to decouple the packet in several DMA/BMM requests), the ping process needs to wait for available space (DMA) or credit (BMM) to continue the data transfer. This difference is less visible for the 64 bytes FIFO due to the overhead generated by request creation at software level. Nonetheless, real case scenarios having one task producing data for eight consumers are not typical. Therefore, in order to keep a compromise between performance and cost, the remaining evaluations employ 128 bytes FIFOs.

To further evaluate the communication time for a fixed FIFO size, a scenario exchanging 128 kB of data between each ping-pong connection is evaluated. The comparison between DMA and BMM performances is depicted in Figure 5.18. The number of connections used in this scenario are 2, 4 and 8, with packet sizes ranging from 8 bytes to 8 kB. The gain curve shows that the communication time can be further decreased, in comparison with DMA, when exchanging large amounts of data (up to 98%). The gains are similar regardless the number of pong processes, showing that BMM can efficiently handle several connections in parallel.

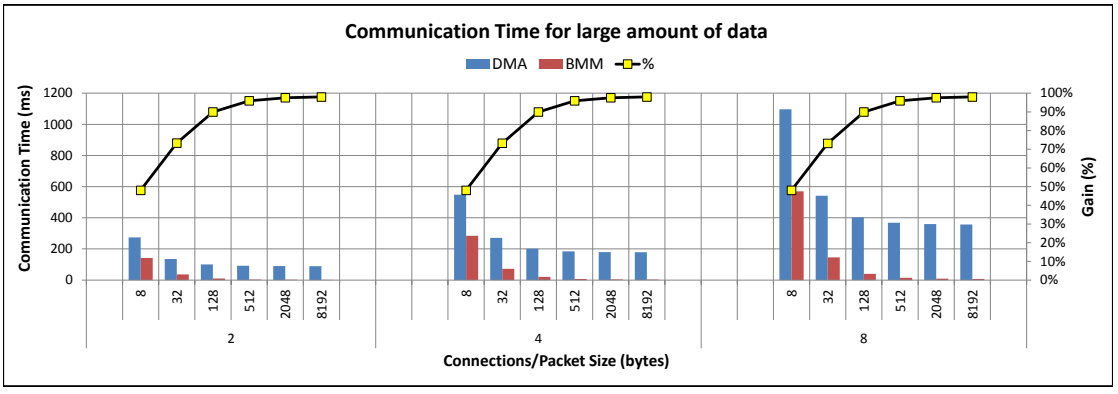


FIGURE 5.18: Communication time comparison for a data transfer of 128 kB.

Finally, the application execution time is evaluated using the same scenarios of Figure 5.18 and the results are presented in Figure 5.19. The communication time dominates the total execution time, since the amount of data exchanged hides channel set-up and initialization times. It is clear that managing FIFO communication in software highly impacts the application performance, even with a DMA being in charge of data transfers. The gains obtained with BMM are lower for smaller packet sizes, since

the application needs to create a higher number of requests. Nevertheless, the gains range from around 50% for a packet size of 8 Bytes to 97% for a packet size of 8 kB.

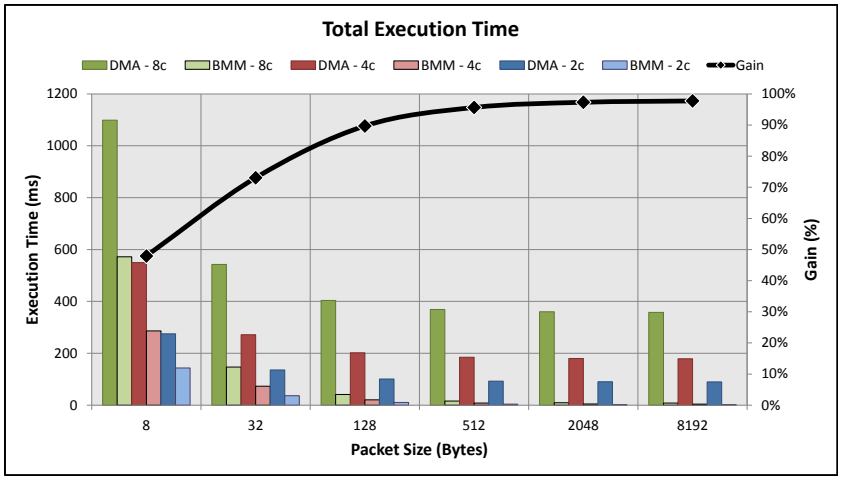


FIGURE 5.19: Ping-pong application execution time to transfer 128 kB of data with different number of connections.

In summary, the characterization of the Buffer Manager Mechanism showed that it can provide very significant gains in terms of throughput, latency, network load, communication time and execution time. While the throughput was increased up to 40 times, the latency and communication time were decreased up to 97%. As the application used to characterize communication time is used only for data transfers, the total execution time presented gains are similar to communication time gains. Therefore, the next section present performance evaluations benchmarks by using benchmark applications.

5.5 Benchmarks Validation

The evaluations using real application benchmarks are used to validate the results obtained with the "*ping-pong*" application and to demonstrate that the mechanisms co-designed with MCAPI fulfill the objectives presented in Chapter 1. Therefore, network load and communication and execution time are evaluated with two benchmarks: *SUSAN* and *Djisktra*. The results obtained with *SUSAN* are presented in Section 5.5.1 and the results obtained for two scenarios with *Djisktra* are presented in Section 5.5.2.

5.5.1 SUSAN

As mentioned in Section 5.1, *SUSAN* (Smallest Univalue Segment Assimilating Nucleus) is a video processing algorithm used for recognizing corners and edges in images. The parameter used to evaluate this application is the number of CPUs processing the input

data, which is a 256x256 pixel image. The algorithm is implemented using a master-slave approach, where all the CPUs run the same algorithm over a part of the image. However, the number of processors used must be known before the application execution.

Initially, the "master" processor, which can be any processor, divides the image in blocks by the amount of processors that will be used to execute the application and sends the data for each "slave" processor. Then, each processor will execute the first part of the algorithm over its respective block. Next, before executing the last part of the algorithm, each "slave" must exchange data with the processors that have adjacent image blocks of its own. Therefore, it sends part of the processed block as well as receives parts of adjacent blocks to complete the algorithm. Finally, all "slaves" return the processed blocks to the "master", which is responsible for writing the data in the output file.

Figure 5.20 shows the average number of flits sent by each slave task. The network traffic for 2 slave tasks is significantly lower due to the application task mapping, which placed one of the tasks in the same cluster/domain as the master task. However, with 4 or more CPUs running a slave task, the results are similar to the gains presented in Section 5.4.3. The gain slightly decreases as the number of CPUs increases, since the amount data exchanged per task is decreased, which in turn, decreases the overhead imposed by the FIFO API and DMA.

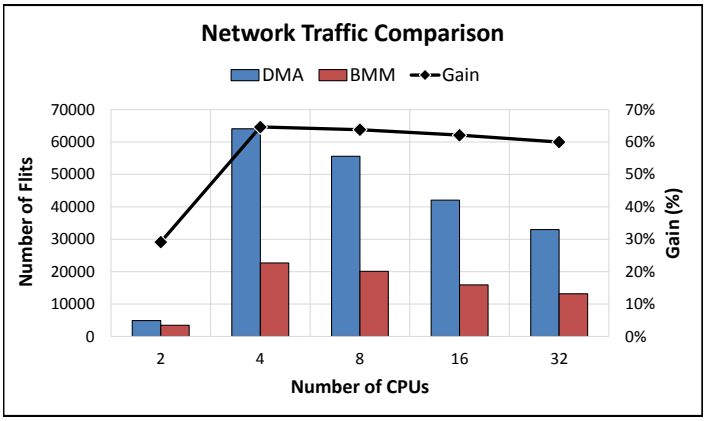


FIGURE 5.20: Average number of flits sent by each task for SUSAN benchmark.

The communication time evaluation is depicted in Figure 5.21. Similarly to the network traffic evaluation, the communication time is showed as an average of the communication time in the slave tasks. The "Gain" curve shows that the communication time was decreased around 95% regardless the number of CPUs executing the benchmark. Since a large amount of data is exchanged in this application (around 500 kB) [71], this benchmark corroborates the results presented in Figure 5.18, showing a very significant decrease in the software implementation overhead.

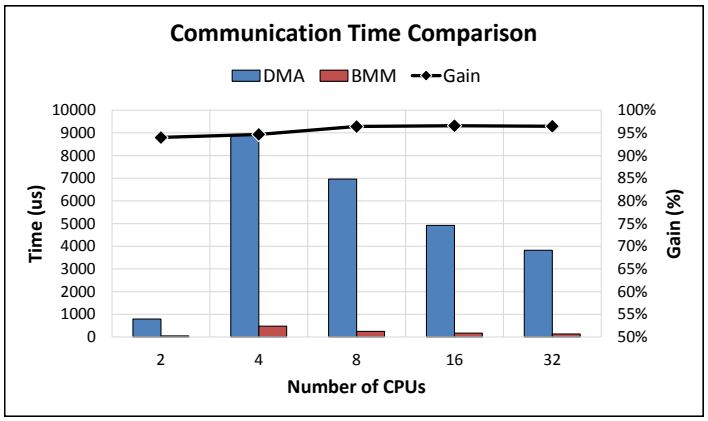
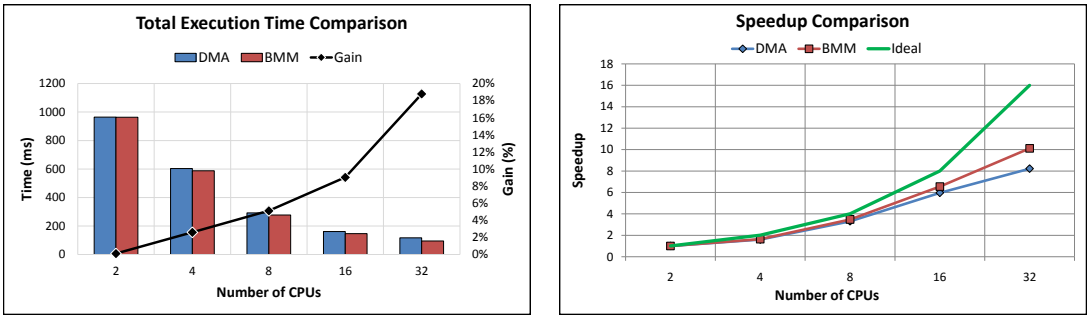


FIGURE 5.21: Average communication time comparison between BMM and DMA for SUSAN benchmark.

Lastly, Figure 5.22 shows the evaluations for total execution time (a) and speedup (b) in the same scenario. It is possible to notice that, with fewer CPUs, the significant decrease in the communication time presented in Figure 5.21 has lower impact in the application performance, since the CPUs spend considerably more time processing than transferring data. However, as the number of CPUs increases, the gain in the application performance also becomes significant, due to the higher impact of the communication time over total execution time. This is also evidenced in Figure 5.22(b), where the speedup with 32 CPUs is around 10 when using the BMM and around 8 when using the DMA and FIFO API.



(a) (b)

FIGURE 5.22: Average total execution time (a) and speedup (b) comparison between BMM and DMA for SUSAN benchmark.

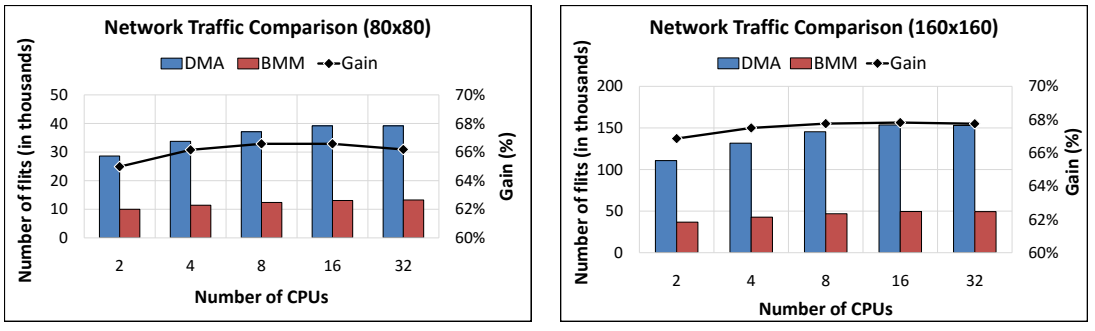
5.5.2 Dijkstra

The second benchmark evaluated is called *dijkstra*, and similarly to SUSAN, the evaluation parameter is the number of processing cores used to execute the algorithm. The benchmark consists on finding the shortest path between every pair of nodes in a given

graph. The shortest path calculation is performed by the Dijkstra's algorithm [72], which is a well known solution to the shortest path problem with run-time complexity of $O(n^2)$.

The application is also implemented using a master-slave approach, where the master process creates a list of tasks that will be issued by the different slave processes. Then, the graph is sent to the Shared Memory of each cluster as a matrix, where the value x stored in the position $\{i,j\}$ corresponds to the distance between the nodes i and j. The evaluation is performed for 2 matrix sizes: 80x80 and 160x160. Next, each slave receives a task from the master, process it and returns the result. If the application is not finished, i.e., there are available tasks, the master sends another task to the slave that returned a result. Otherwise, it sends a message signaling that the application has finished. In this benchmark, the slave processes do not exchange messages.

Figures 5.23, 5.24 and 5.25 present the results regarding network traffic, communication and execution times, respectively. All evaluations depict the results for the small scenario (80x80) in Figure (a) and the results for the big scenario (160x160) in Figure (b). The gains in terms of network traffic are similar for both scenarios and also similar to the previous results regarding network traffic, as showed in Figure 5.23. In average, the decrease in number of flits sent by all processors is of 66% and 67.5%, respectively. This is explained by the fact that the largest data transfer occurs at the application start, where the graph is sent to all processors as a single packet. Thus, the average decrease is around the values presented for large packets in Figure 5.16.

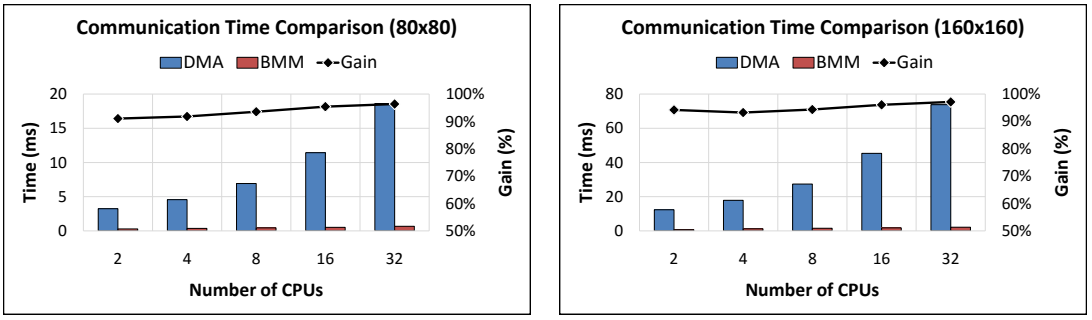


(a) (b)

FIGURE 5.23: Average number of flits sent by each task for Dijkstra benchmark. (a) Scenario with 80x80 matrix. (b) Scenario with 160x160 matrix.

Figures 5.24 and 5.25 show the average communication and execution times, respectively, between the processors of a given scenario. It is possible to note that, in Figures 5.24(a) and 5.24(b), the highest communication time using the DMA is around 2 and 8 times higher when compared to *SUSAN* benchmark, respectively. Nonetheless, the average gains slightly change, staying always higher than 93%. These results show

that the BMM can handle different traffic patterns and communication volume without performance degradation.



(a) (b)

FIGURE 5.24: Average communication time comparison between BMM and DMA for Dijkstra benchmark. (a) Scenario with 80x80 matrix. (b) Scenario with 160x160 matrix.

Finally, the gains in the application performance are measured through the total execution time and speedup, which are presented in Figure 5.25. Similarly to *SUSAN* benchmark, the gain curve shows that the performance increase is higher when the communication time has higher impact over total execution time. However, Figure 5.25(a) achieves higher gains than Figure 5.21 (up to 26%) since, in contrast to *SUSAN*, the communication time increases proportionally to the processor count. Figure 5.25(b) corroborates this behavior, presenting lower gains than *SUSAN*.

This is also noticed in Figures 5.25(c) and 5.25(d), where the speedup difference between the two approaches is higher in the 80x80 matrix scenario. Nevertheless, although the gains were decreased by one-third compared to the small scenario, the execution times are around 15 times higher. Therefore, the BMM offers significant performance gains for applications that are not computation-bound and/or for architectures with a high number of processing cores.

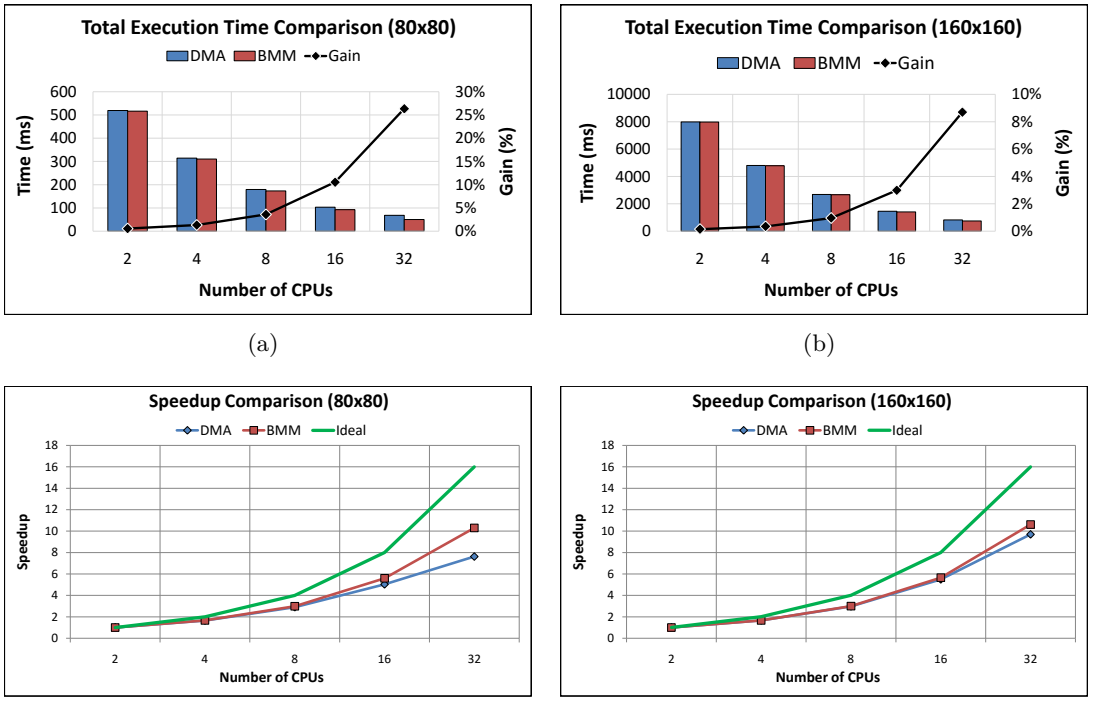


FIGURE 5.25: Average total execution time and speedup comparison between BMM and DMA for Dijkstra benchmark. (a) Total execution time in the scenario with a 80x80 matrix. (b) Total execution time in the scenario with a 160x160 matrix. (c) Speedup in the scenario with a 80x80 matrix. (d) Speedup in the scenario with a 160x160 matrix.

Conclusion and Perspectives

Programming distributed and parallel applications in embedded multi-core systems have become increasingly complex due to the system heterogeneity. Additionally, the lack of standard software solutions contributes to further increase this complexity. Thus, in order to decrease the gap between hardware and software technologies, this Thesis has addressed hardware and software co-design of multi-core architectures. The objective was to increase programmability through the implementation of a software standard for inter-process communication while decreasing the performance overhead imposed by this implementation and, hence, improving communication performance. Three main contributions were developed to achieve this:

- Implementation of MCAPI for a generic multi-core architecture (Chapter 2).
- Implementation and evaluation of a hardware mechanism for communication set-up and synchronization (Chapter 3)
- Implementation and evaluation of a hardware mechanism for data transfers (Chapter 4)

The programmability of multi-core architectures is improved by the implementation of MCAPI, which specifies primitives for inter-process communication. Then, the overheads imposed by the software implementation were characterized and mechanisms addressing communication set-up and data transfer were proposed.

Regarding the MCAPI implementation, it is showed that the API is lightweight and compatible for embedded systems. The implementation performed in the scope of this Thesis reported to be smaller than the implementations performed by other works. However, it is important to highlight that the total memory footprint, i.e., considering the data structures, depends also on the application constraints. For instance, the less the number of simultaneous connections a single node requires, the lower is the total memory footprint.

The first overheads identified in the software implementation were increased network traffic and processing loads during the communication set-up phase. These overheads are caused by the polling processes performed at each connection set-up steps. To overcome this issue, the Event Synchronizer (ES) hardware module is proposed. It works as a programmable mechanism able to handle a parameterizable number of events. As showed in Chapter 3, this mechanism is easily accessible and programmable since it was developed in co-design with MCAPI. The results obtained when taking advantage of the ES are presented in Section 5.3, and show that both network traffic and CPU loads are significantly decreased, up to 87% and 88% respectively, without increasing software complexity.

Next, the overheads presented in data transfer phase were evaluated, showing that the FIFO control implementation in software also induces increased network traffic and processing loads. Therefore, the Buffer Manager Mechanism (BMM) is proposed in order to implement FIFO control in hardware. Also, in order to achieve higher flexibility, the mechanism does not provide FIFO structures or impose constraint about memory sizes. Instead, it has table structures to store FIFO parameters, such FIFO sizes and their placement. Additionally, it uses a credit-based flow control to avoid pointer exchanges. Moreover, the BMM can be programmed through memory-mapped registers, allowing it to be used by hardware accelerators and, similarly to the ES, not increasing software complexity. The characterization regarding latency, throughput, network and application performance is presented in Section 5.4 and shows significant gains: throughput increased up to 30 times, latency decreased up to 97%, network traffic decreased up to 68% and total execution time decreased up to 97%.

In addition to the mechanisms characterization, which was performed with the "ping-pong" benchmark, their performance was evaluated using real applications. The results presented in Section 5.5 corroborate the gains obtained in the previous evaluation regarding network traffic and communication time. However, it is demonstrated that with applications that are not computation bound, the gains in terms of total execution time vary from 1% to 26%. As conclusion, the proposed mechanisms are more effective for communication bound applications or when employed by a high number of processing engines.

Therefore, this Thesis has demonstrated that co-designing hardware mechanisms and MCAPI can significantly decrease the performance overheads imposed by software implementation. Also, it is showed that the mechanisms can be flexible and easily programmable, which does not increase programmability complexity or memory footprint.

Future Works

The contributions made in this Thesis are a first step in the hardware and standard software co-design for embedded systems, and several works can be envisaged based on them. A natural addition is to exploit the proposed mechanisms for the other two communication modes supported by MCAPI: messages and scalar channels. This extension is performed at the API transport level and does not require any hardware modification. Also, the number of System Buffers can be increased in order to allow multiple reception flows in a cluster and/or node. However, this improvement would require a hardware mechanism to handle the resources and possibly implement dynamic memory allocation, since the System Buffers are a finite resource and the applications may call the receive primitive in a different order of the received data. Another possible solution is to implement the zero-copy concept, which would allow the data to be transferred to the application directly from the FIFOs and, consequently, increase the communication performance. Finally, regarding the MCAPI implementation, the receiver-initiated messages concept ([73]) can be evaluated and compared with the adopted pre-pushing strategy.

The perspectives about the mechanisms are to increase their functionality and flexibility to support communication and/or synchronization operations that are not currently supported by MCAPI. Thus, other APIs targeting different aspects, such as MRAPI (resource management) and MTAPI (task management) can also take benefit from the proposed mechanisms. Furthermore, according to the MCAPI website [6], a new specification is under development, requiring the mechanisms to be updated accordingly.

In terms of performance evaluation, additional benchmarks with different relations between computation and communications can be considered. Furthermore, different architecture configurations might change the results obtained with *susan* and *dijkstra*, e.g., the processor operating frequency higher than NoC operating frequency. In this case, the performance increase is expected to be greater, i.e., higher decrease in total execution times.

Finally, the mechanisms can be inserted in a heterogeneous platform to demonstrate their flexibility. As the mechanisms are programmable through memory-mapped registers, hardware accelerators might interpret them as memory extension. Furthermore, by using MCAPI, the application can abstract the platform heterogeneity, leaving to the transport layer implementation the responsibility of programming the hardware accordingly. Thus, the software component responsible for controlling the respective hardware accelerator has to be modified in order to access the mechanisms. However, as already demonstrated in this Thesis, the software complexity is not increased.

Appendix A

List of Publications

1 Patent:

Thiago Raupp da Rosa, Romain Lemaire, Fabien Clermidy. "**Système sur puce et procédé d'échange de données entre nœuds de calculs d'un tel système sur puce**", France, Number 1652043. Filling date: March 11th, 2016.

2 Conference Papers:

Da Rosa, T.R.; Mesquida, T.; Lemaire, R.; Clermidy, F., "**MCAPI-compliant Hardware Buffer Manager Mechanism to Support Communication in Multi-Core Architectures**," in Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016 , vol., no., pp., 14-18 March 2016

Da Rosa, T.R.; Lemaire, R.; Clermidy, F., "**A Co-design Approach for Hardware Optimizations in Multicore Architectures Using MCAPI**," in Ninth International Workshop on Interconnection Network Architectures: On-Chip, Multi-Chip (INA-OCMC), 2015, vol., no., pp.17-20, 19-19 Jan. 2015

Appendix B

Résumé en Français

B.1 Introduction

Les progrès technologiques ont permis jusqu'à présent l'intégration toujours croissante d'un plus grand nombre de composants sur une seule puce [1], nommée «SoC» (système-sur-puce). Au cours des dernières décennies, conjointement à l'augmentation du nombre de transistors, la performance des SoC pouvait être améliorée en augmentant également leurs fréquences de fonctionnement. Cependant, ces changements d'échelle atteignent des limites, telles que le «power wall» [2], cette approche ne peut être reproduite indéfiniment. Ainsi, pour augmenter davantage les performances des systèmes embarqués, des architectures utilisant plusieurs cœurs de traitement ont été largement utilisées au cours des dernières années.

Actuellement, en plus de l'exigence de performance plus élevée, plusieurs domaines applicatifs (tels que les télécommunications haut-débit, la vision artificielle ou le traitement vidéo haute-définition) imposent également des consommations d'énergie réduite comme une contrainte primaire. Par conséquent, des accélérateurs matériels peuvent être utilisés pour atteindre une meilleure efficacité énergétique. Dans ce contexte, développer des applications embarquées sur des plateformes hétérogènes devient une problématique complexe. En effet, des difficultés de programmation apparaissent compte-tenu de l'absence de standards logiciels qui prennent en charge l'hétérogénéité des nouvelles architectures, menant souvent à des solutions adhoc.

Une API (Interface de Programmation Applicative) de programmation standardisée pour les systèmes embarqués est proposé par la «Multicore Association» (MCA) [6]. La «Multicore Communication API» (MCAPI) est mise en œuvre dans [10–12]. Cependant, comme le montrent ces travaux, l'utilisation de MCAPI peut induire des surcoûts

importants concernant les performances. L'ajout de mécanismes matériels pour gérer les communications inter-processus peut augmenter la performance globale en accélérant les phases de communication. D'autre part, dans la plupart du temps, ces mécanismes ne sont pas flexibles, en ce qui concerne leur utilisation dans différentes architectures, ou ne prennent pas en compte la complexité accrue dans le développement de logiciels pour les gérer. En conséquence, il apparait pertinent de suivre une approche de co-conception logicielle et matérielle afin d'augmenter la programmabilité des architectures multi-cœur tout en répondant aux exigences de performances. Néanmoins, l'utilisation d'une API (Interface de programmation) logicielle standard est essentielle pour la réutilisation et compatibilité du code applicatif.

Les principaux objectifs sont ainsi d'accroître la programmabilité et de diminuer la charge logiciel des cœurs de processeur grâce à l'utilisation de mécanismes matériel adéquates. Afin d'atteindre ces objectives, les trois contributions principales de cette thèse sont:

- La mise en œuvre et l'évaluation d'une Interface de programmation (API) standard pour les communications inter-processus.
- La conception et évaluation d'un mécanisme matériel pour améliorer la performance dans la phase de synchronisation de la communication.
- La conception et évaluation d'un mécanisme matériel pour améliorer la performance des transferts de données.

B.1.1 Organisation de la Thèse

Le résumé est divisée en sept sections. L'introduction décrit le contexte de ce travail et présente ses contributions. La Section B.2 place la thèse en relation avec l'état de l'art et décrit l'architecture de référence. La Section B.3 présente le standard MCAPI et les choix de conception effectués pour sa mise en œuvre sur l'architecture de référence. Une analyse est également effectuée pour évaluer les principaux surcoûts de cette mise en œuvre.

Ensuite, la Section B.4 présente le mécanisme «Event Synchronizer», qui est la deuxième contribution de cette thèse. Puis, la Section B.5 présente la troisième contribution de cette thèse, qui est le mécanisme «Buffer Manager». L'environnement de simulation et les résultats expérimentaux sont décrits dans la Section B.6. Finalement, la conclusion est présentée dans la Section B.7.

B.2 Vue d'ensemble des Systèmes Multi-cœurs

Les architectures multi-cœurs ont été utilisées au cours des dernières années en tant que solution pour répondre aux contraintes de performance dans plusieurs domaines applicatifs. Par ailleurs, selon l'ITRS [13], le nombre de cœurs de traitement intégrés sur une seule puce devrait croître de façon exponentielle dans les prochaines années, ce qui confirme l'importance de développer davantage ces architectures. Au-delà du nombre de cœurs de traitement, leurs types doivent également être pris en considération. Lorsque des cœurs de types différents sont utilisés sur une même puce, l'architecture peut être classée comme hétérogène. A l'inverse, les architectures homogènes sont composées d'un seul type de cœur dupliqué, qui peut être soit un processeurs générique (par exemple, les processeurs multi-cœurs d'Intel, SCC [14]) soit un processeur DSP [15]. Les deux types d'architectures homogènes ou hétérogènes ont leurs avantages et inconvénients. Même si les applications sont plus faciles à coder et paralléliser sur les architectures homogènes, l'efficacité énergétique des solutions hétérogènes est plus élevée ce qui est souvent un critère prépondérant. Le Tableau B.1 montre la comparaison entre plusieurs architectures en ce qui concerne leurs applications cibles (spécifique, semi-spécifique ou générique), la communication inter-processus (IPC), le contrôle de la communication, les caractéristiques matérielles (interconnexion et de traitement) et le support pour la programmation de l'application.

Les applications cibles des architectures sont multiples et une tendance ne peut pas être identifiée. Cependant, avec les dispositifs mobiles supportant de multiples fonctionnalités (normes 4G et 5G, lecture vidéo de haute-défintion, etc) et l'Internet des objets (IoT), il devient de plus en plus en évident que des architectures ciblant des niches spécifiques d'application ne sont pas suffisantes. Dans la troisième colonne, les communications inter-processus sont principalement réalisées par le biais de mémoires partagées avec un contrôle de centralisé. Cependant, avec l'augmentation du nombre de nœuds de traitement, les architectures doivent mettre en œuvre un contrôle distribué pour éviter les goulots d'étranglement dans le système. Par conséquent, l'architecture multi-cœur utilisée comme référence mettra en avant la flexibilité et la scalabilité comme les principales contraintes cibles.

B.2.1 Architecture de Référence

L'architecture de référence est composée de plusieurs clusters connectés par un réseau sur puce (NoC). Chaque cluster comprend plusieurs processeurs (MIPS R3000) avec leurs mémoires privées respectives, des modules de sortie et d'entrée (sous-système CPU),

TABLEAU B.1: Résumé et comparaison des architectures multi-cœurs.

Architecture	**Application Cible**	**IPC**	**Contrôle de la Communication**	**Infrastructure de Interconnexion**	**Principaux Cœurs de Traitement**	**Accélérateurs Matériels**	**Programmation de l'Application**
big.LITTLE [16]	Générique	Mémoire Partagée	Centralisé	Interconnexion Propriétaire	Processeurs basé sur ARM	-	-
Cell [17]	Générique	Mémoire Partagée	Centralisé	des bus en anneau	Power PC	Unités de Traitement Synergiques	-
Tomahawk [19]	Semi-Spécifique	Mémoire Partagée	Centralisé	basée sur NoC	Tensilica DC212	VDSPs, SDSPs, Décodeur LDPC, etc	basée dans C; #pragmas
Faust [23]/Magali [20]	Spécifique	Passage de Messages	Partiellement Distribué	basée sur NoC	Processeurs basé sur ARM	FFT, IFFT, VLIW DSPs, etc	Configuration de flux de données
X-GOLD SDR20 [21]	Spécifique	Mémoire Partagée	Centralisé	basée sur Bus	Processeurs basé sur ARM	Processeurs SIMD, Audio DSP, etc	-
COBRA [22]	Spécifique	Mémoire Partagée	Centralisé	des crossbars et des bus	ADRES	DIFFS, FlexFEC, Accélérateur Viterbi	Outil MPA
Flextiles [26]	Générique	Mémoire Partagée et Passage de Messages	Distribué	basée sur NoC	GPPs	DSPs, eFPGA	basée dans threads; langages spécifiques
P2012 [27]	Générique	Mémoire Partagée	Partiellement Distribué	basée sur NoC	STxP70-V4	HWPEs	Plusieurs options

une mémoire partagée, une interface réseau (NI) et un DMA, comme représenté sur la Figure B.1. Le sous-système CPU envoie et reçoit des messages de contrôle à travers des modules de sortie et d'entrée, respectivement. Le DMA est capable d'effectuer des transferts de données par des requêtes émises par les processeurs. Ces requêtes fournissent des paramètres habituels: adresse du buffer d'origine, adresse du buffer de destination, la taille du transfert et l'identification de transfert (ID).

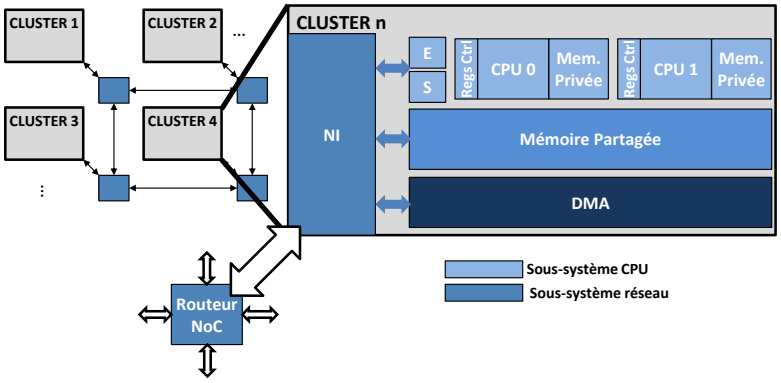


FIGURE B.1: Schéma et hiérarchie de l'architecture de référence.

Du point de vue logiciel, un ensemble de fonctions est prévue pour la communication inter-processus. La communication est gérée par FIFO qui sont placées dans la mémoire partagée et contrôlées par des routines logicielles. Ainsi, l'application n'a pas besoin de gérer les adresses locales ou distantes dans les transferts de données, ce qui réduit la complexité de la programmation. En outre, comme l'adressage est global, l'ensemble de l'espace mémoire peut être considéré comme une seule mémoire partagée distribuée avec des temps d'accès non uniformes (NUMA), ce qui rend possible pour tout CPU d'accéder dans n'importe quelle mémoire FIFO quelle que soit son placement.

B.2.2 Positionnement de la Thèse

MCAPI cible les communications inter-processus et les travaux de recherche doivent s'orienter sur des solutions qui améliorent cet aspect. Cette thèse se concentre sur la mise en œuvre du mode de communication par canaux de paquets décrit par le standard MCAPI. Ceci peut être expliqué par le fait qu'ils sont plus flexibles que les canaux de scalaires et offrent de meilleures performances par rapport aux messages, couvrant une large gamme d'applications de flux de données. Pour utiliser ce type de canal, il est obligatoire d'abord établir une connexion.

En effet, des actions spécifiques doivent être exécutées par les deux extrémités de la communication avant et après l'échange de données. Donc, le processus de communication peut être divisé en deux phases: la configuration de la communication (synchronisation) et le transfert de données. La première phase est utilisée pour établir une connexion et pour allouer et désallouer des ressources alors que la seconde est utilisée pour échanger des données. Par conséquent, les deux phases doivent être prises en compte lors de l'élaboration des mécanismes matériels. Le résumé de cet étude est présenté dans le Tableau B.2.

TABLEAU B.2: Positionnement de la thèse concernant les aspects de communication et de programmabilité en relation avec l'état de l'art.

Référence	**Flexibilité**	**Support de API SW**	**Support HW pour la transfert de données**	**Support HW pour la synchronisation**	**Architecture Cible**
Calcado [49]	+	Customisé	Non	Oui	Systèmes embarqués étroitement couplé
Kachris [50]	++	Pas Mentionné	Oui	Oui	Systèmes embarqués
Meyer [51]	+	MPI	Non	Oui	Clusters SMP
Tabhet [52]	+++	RTM API	Non	Oui	P2012 [27]
Reble [53]	+++	MPI Customisé	Non	Oui	SCC [14]
Kim [54]	+	Customisé	Non	Oui	Systèmes embarqués
Papadopoulos [55]	+	Pas Mentionné	Non	Oui	Systèmes embarqués
Han [56]	+	Customisé	Oui	Non	Systèmes embarqués
Buono [57]	+	Customisé	Oui	Non	Multi-cœur Multi-threaded
Gao [58]	++	Pas Mentionné	Oui	Non	Clusters Hétérogènes
Kumar [59]	++	Customisé	Oui	Non	Systèmes embarqués
Wallentowitz [60]	+++	MCAPI	Oui	Non	Systèmes embarqués
Clermidy [61]	++	Customisé	Oui	Oui	Systèmes embarqués
Helmstetter [62]	+++	Pas Mentionné	Oui	Oui	Systèmes embarqués
Burgio [63, 64]	+	Customisé/ Open MP	Oui	Non	Systèmes embarqués étroitement couplé
Ku [65]	+	Customisé	Oui	Non	Systèmes embarqués
This thesis	+++	MCAPI	Oui	Oui	Systèmes embarqués

Ainsi, le travail présenté dans cette thèse diffère des travaux antérieurs en fournissant un support matériel flexible pour la configuration de la communication ainsi que pour les transferts de données, conçu conjointement avec une API de communication standardisée.

B.3 Mise en Œuvre de MCAPI et Caractérisation des Surcoûts

MCAPI est une API spécifiée par le Multicore Association [6] qui définit des fonctions pour la communication et la synchronisation inter-processus dans les systèmes embarqués. Son principal objectif est de fournir de la portabilité pour des codes d'application, des performances scalables de communication et une faible empreinte mémoire. La spécification MCAPI définit deux niveaux de hiérarchie: les domaines et les nœuds. Un domaine est composé d'un ou plusieurs nœuds. Un nœud est défini comme un *thread* (fil d'exécution) indépendant de contrôle, i.e. une entité qui peut exécuter uniquement un flux séquentiel d'instructions, à savoir une tâche, un processeur, un accélérateur matériel, etc. Comme l'architecture de référence est divisée en clusters, chacun représente un domaine, alors que chaque CPU à l'intérieur d'un cluster représente un nœud (Figure B.2).

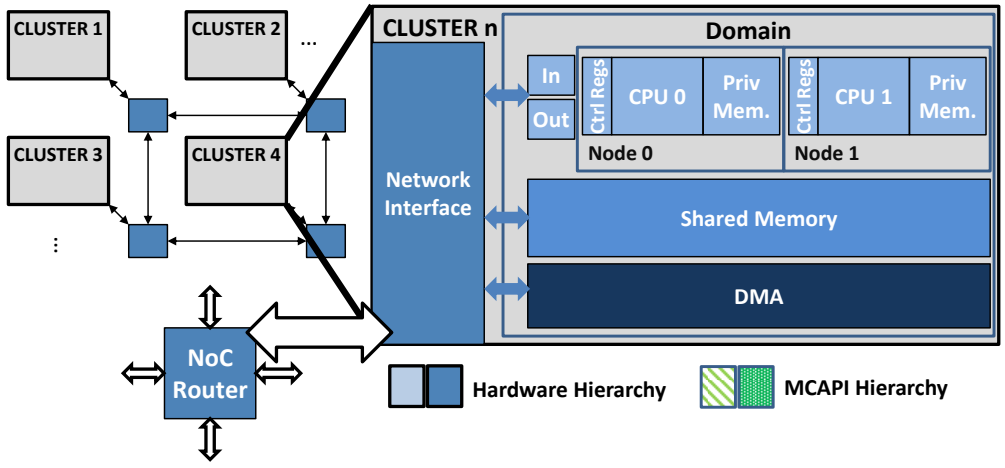


FIGURE B.2: Mappage des domaines et des nœuds MCAPI dans l'architecture de référence.

La communication est établie entre les nœuds à travers une paire de «endpoints». Trois modes de communication sont disponibles: des messages, des canaux de paquets et des canaux de scalaires. La principale différence entre les messages et les canaux (paquets ou scalaires) est la flexibilité, ce qui permet la transmission de données entre les nœuds sans établir une connexion. D'autre part, les canaux offrent une performance supérieure en raison des ressources déjà allouées dans la phase de synchronisation. Les canaux transmettent des données d'une façon FIFO. La différence entre les paquets et les scalaires concerne la taille de transferts de données, quand les canaux de paquets peuvent transférer des blocs de données de tailles variables, les canaux scalaires transfèrent seulement de données de tailles fixes (8, 16, 32 ou 64 bits). De plus, une zone de la

mémoire partagée est allouée de manière statique pour stocker trois types de structures de données: attributs MCAPI, FIFOs et requêtes.

B.3.1 Limitations de Performance

Le principal avantage de la mise en œuvre de MCAPI en logiciel est la flexibilité. A l'opposé, cette mise en œuvre peut induire des surcoûts importants. Afin d'éviter ces surcoûts, la mise en œuvre de MCAPI a été analysée à 2 niveaux: la synchronisation et les transferts de données. Dans le premier point, il a été identifié que les phases d'attente active (*polling*) effectuées pour vérifier les attributs du endpoint induisent des surcoûts de trafic réseau et de charge processeur. En ce qui concerne le deuxième point, les mêmes surcoûts sont induits par le contrôle FIFO implémenté en logiciel. Par conséquent, deux mécanismes ont été développés pour résoudre les limitations identifiées.

B.4 Support pour la Configuration de la Communication

Afin de résoudre les problèmes mentionnés ci-dessus, un mécanisme appelé «Event Synchronizer» (ES) est proposé. Ce module a comme objectif d'être flexible tel que les solutions présentées dans [52] et [53], et aussi d'être développé en co-conception avec MCAPI. L'ES est un module matériel programmable capable de gérer un nombre paramétrable d'événements. De plus, pour une plus grande flexibilité, chaque CPU est attaché à un module indépendant.

L'ES est composé de multiples registres de synchronisation d'événements (SERs), de registres d'identification de connexions distantes (RCRs), de deux registres de masque et de 4 processus pour traiter les informations reçus et les événements générés. Les SERs sont chargés de stocker les événements pour chaque terminal de communication (*endpoint*). Les RCRs sont des registres de 32 bits utilisés pour stocker les identifiants de connexion distante pour chaque endpoint local. Les registres de masque sont utilisés pour programmer les événements attendus par le processeur. L'accès se fait grâce à des registres visible dans l'espace d'adressage, où le processeur peut écrire et lire les masques.

Ainsi, en utilisant l'Event Synchronizer, il est possible de supprimer toutes les pollings effectués lors de la synchronisation des deux côtés de la communication. Ceci est réalisé par la définition d'un événement différent pour chaque phase de synchronisation. La Figure B.3 montre comment l'ES est utilisé (à droite) par rapport à la mise en œuvre de MCAPI en logiciel (à gauche). En plus d'éviter le trafic réseau inutile, l'ES permet

au CPU d'entrer dans un état "inactif", qui peut être traduit dans un état de faible consommation.

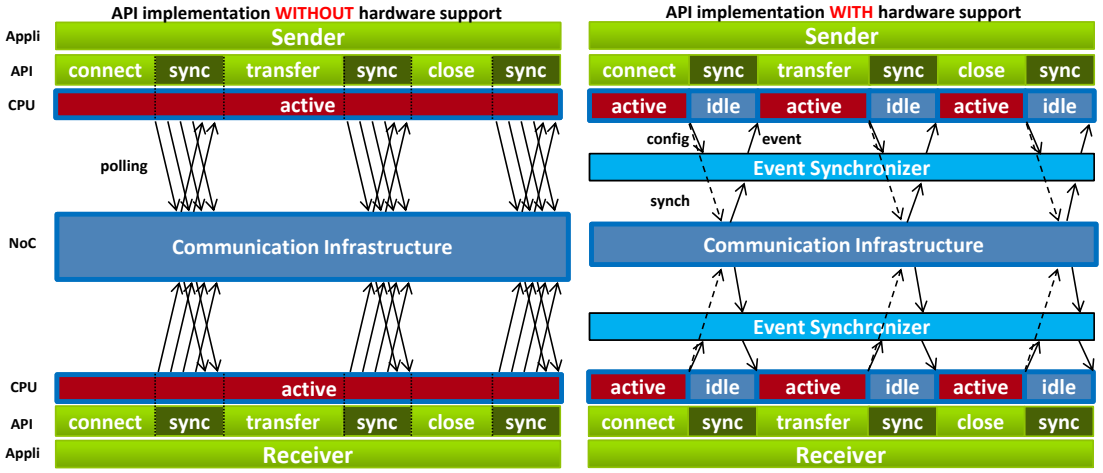


FIGURE B.3: Comparaison d'utilisation de la plate-forme matériel sans et avec l'ES.

B.5 Support pour les Transferts de Données

Le mécanisme proposé est appelé «Buffer Manager Mechanism» (BMM) et a comme objectifs réduire les surcoûts de trafic réseau et de charge processeur pendant la communication inter-processus. Il est composé de 4 modules matériels: BMI, BMW, BMR et CM. Ce mécanisme a été développé en co-conception avec MCAPI et remplace le DMA dans les transferts de données. Les principales différences entre le BMM et le DMA sont que le BMM peut gérer des opérations de lecture et d'écriture ainsi que l'utilisation d'identificateurs de connexion. De plus, en raison de ces identificateurs de connexion, le BMM est capable de gérer plusieurs transferts de données en parallèle en utilisant des requêtes de lecture et d'écriture, ce qui réduit le coût du matériel tout en augmentant la flexibilité.

Le BMI (Buffer Manager Interface) est responsable de la lecture et de la paquétisation des données à transmettre du côté émetteur, alors que le BMW (Buffer Manager Write) est responsable, du côté récepteur, de la réception et de l'écriture des données dans les mémoires FIFO respectives. Le BMR (Buffer Manager Read) est utilisé du côté récepteur pour récupérer les données reçues. Enfin, le CM (Credit Manager) met en œuvre un contrôle de flux de communication par crédits. Le CM est responsable de l'envoi et de la mise à jour des crédits lorsque les données sont lues ou envoyées. En plus des modules matériels, le BMM utilise trois table pour faire le contrôle de la communication: la table de connexion, la table de crédits et la table des buffers.

Le CPU effectue des opérations d'envoi et de réception de données en créant des requêtes au BMM. Pour créer une requête, les données doivent être écrites dans des adresses spécifiques qui codent les paramètres de la requête, nommées adresses de configuration. Lors de l'envoi des données, trois options sont disponibles: (*i*) transfert basé sur les adresses, (*ii*) transfert *streaming* direct, ou (*iii*) transfert *streaming* indirect. Le transfert à base d'adresses est implémentée comme une fonctionnalité héritée pour effectuer des transferts précédemment réalisé par le DMA. D'autre part, les transferts streaming utilisent les identificateurs de connexion au lieu d'adresses de destination. Dans le transfert streaming direct un seul mot de 32 bits est transmis à partir d'un identificateur source à un identificateur destination, alors que dans le transfert streaming indirect un buffer de mots de 32 bits est transmis à partir d'un identificateur source à un identificateur destination.

La Figure B.4 montre le fonctionnement du mécanisme des deux côtés de la communication lors d'un transfert streaming indirect. Tout d'abord, il y a une phase d'initialisation des deux côtés (*action 1*), qui se compose de l'initialisation de la table de connexions par chaque CPU. Ensuite, au niveau du cluster émetteur, le CPU écrit les données dans un buffer alloué dans la mémoire privée et crée une requêtes d'écriture pour le BMI (*action 2*). Ensuite, le BMI prend la requête et récupère l'identificateur de connexion distant et la quantité crédit disponible (*action 3*). Dans le cas où les crédits sont disponibles, le paquet de données est envoyé (*action 4*) avec les informations de l'identificateur de connexion de destination, et le CM est notifié qu'une mise à jour des crédits est nécessaire.

Dans le cluster récepteur, les données sont reçues par le BMW, qui accède au table des buffers (*action 5*) et écrit les données dans le buffer cible (*action 6*). A un moment donné, la CPU du cluster récepteur crée une requête de lecture pour le BMR (*action 7*). Ensuite, le BMR accède à la table de buffers (*action* 8) et copie les données vers l'adresse de destination (*action 9*). Enfin, le BMR informe le CM qu'il doit générer des crédits pour l'identificateur de connexion distant (*action* 10), qui, à son tour, crée et envoie le paquet de crédit via le NI (*action 11*).

B.6 Résultats Expérimentaux et Validation

Les résultats présentés dans cette section ont été obtenues avec des simulations en utilisant un modèle SystemC [66] de l'architecture de référence (SectionB.2.1), qui a été développé au cours de cette thèse. Le modèle est décrit au niveau TLM (*Transaction Level-Modeling*), les CPU sont modélisés par ISS du MIPS R3000 [67]. Les modules sont connectés via des *sockets* et échangent des transactions TLM génériques [66]. Au

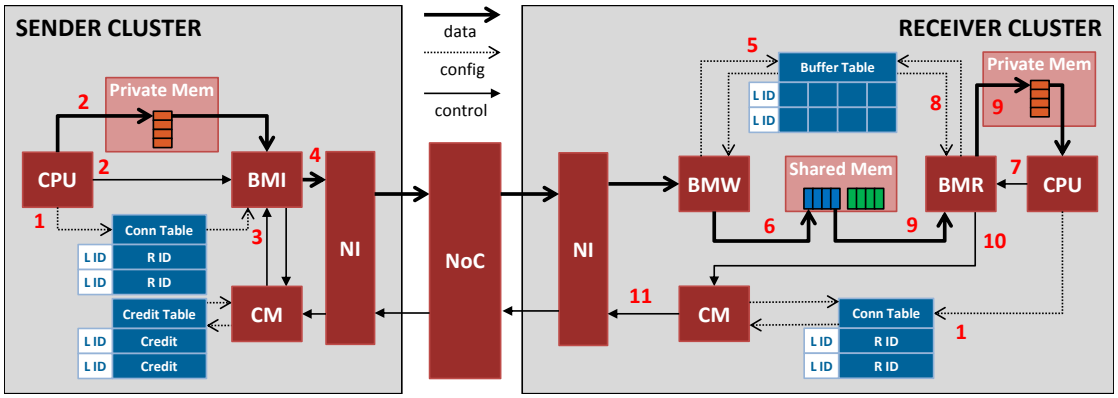


FIGURE B.4: Opération du mécanisme de communication dans une requête streaming indirect.

niveau du réseau, chaque flit est représenté par une seule transaction TLM. Dans les clusters, les transactions sont gérées par un bus générique, qui transmet les transactions en fonction de leurs adresses de destination. Enfin, l'application SUSAN a été utilisées pour valider les gains de performance obtenus avec les mécanismes proposés.

B.6.1 SUSAN

L'application SUSAN (*Smallest Univalue Segment Assimilating Nucleus*) est un algorithme de traitement vidéo utilisé pour reconnaître les coins et les bords des images. Le paramètre utilisé pour évaluer cette application est le nombre de CPU qui traitent les données d'entrée (une image de 256x256 pixels). L'algorithme est implémenté en utilisant une approche maître-esclave, où tous les processeurs exécutent le même algorithme sur une partie de l'image.

La figure B.5 montre le nombre moyen de flits envoyé par chaque tâche esclave. Le trafic réseau pour 2 tâches esclaves est nettement plus faible car l'une des tâches est positionnée dans le même cluster que la tâche maître. Pourtant, avec 4 ou plusieurs processeurs exécutant une tâche esclave, les résultats sont similaires aux gains présentés dans la Section 5.4.3 (manuscrit en anglais): entre 60% et 70%. Le gain diminue légèrement à mesure que le nombre de CPU augmente, étant donné que la quantité de données échangées par tâche est diminuée, ce qui, à son tour, diminue les surcoûts imposée par l'API FIFO logiciel et le DMA.

Le temps de communication est évalué et représenté dans la Figure B.6. Comme pour l'évaluation du trafic du réseau, le temps de communication est calculé comme la moyenne du temps de communication dans les tâches esclaves. La courbe de gain montre que le temps de communication a diminué d'environ 95% quel que soit le nombre de

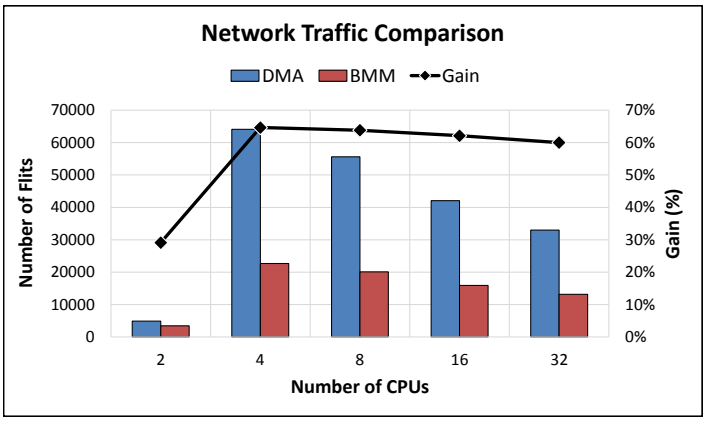


FIGURE B.5: Nombre moyen de flits envoyés par chaque tâche pour l'application SUSAN.

processeurs exécutant l'application. Étant donnée qu'une grande quantité de données sont échangées (environ 500 kB) [71], ce benchmark corrobore les résultats présentés dans la Figure 5.18 (manuscrit en anglais), montrant une diminution très significative des surcoûts de la mise en œuvre en logiciel.

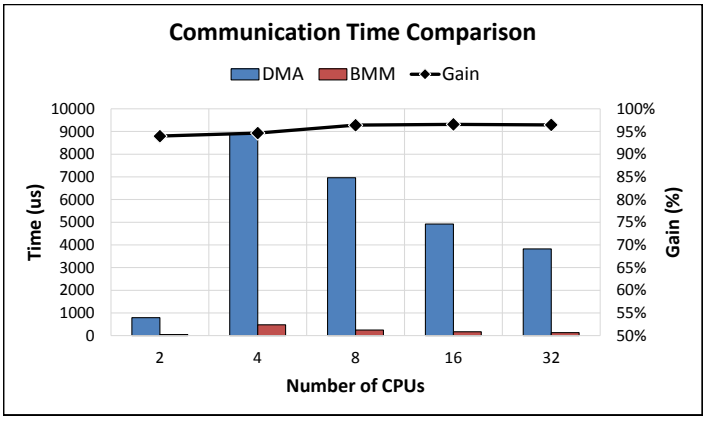
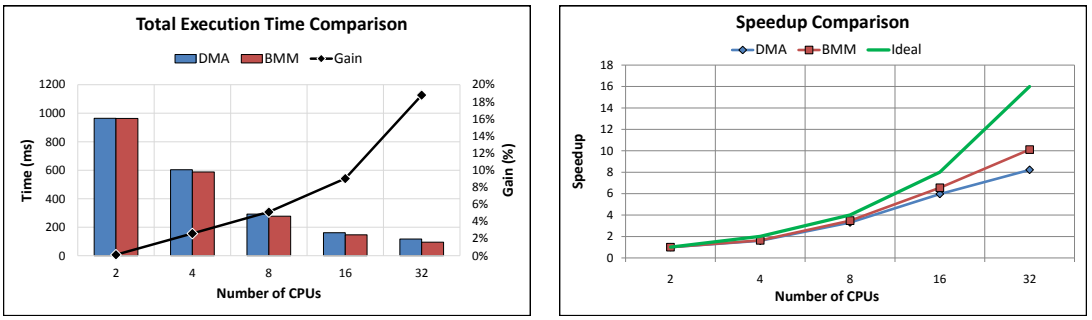


FIGURE B.6: Comparaison des temps de communication moyen entre le BMM et le DMA pour l'application SUSAN.

Enfin, la Figure B.7 montre les évaluations pour le temps total d'exécution (a) et facteur d'accélération (b). Il est possible de constater que, avec moins de CPUs, la diminution significative du temps de communication présentée dans la Figure B.6 a un impact plus faible dans la performance de l'application, puisque les processeurs passent beaucoup plus de temps pour traiter les données que pour les transférer. D'autre part, avec un plus grand nombre de processeurs, le gain de performance devient important, étant donné que le temps de communication a plus d'influence dans le temps total d'exécution. Ceci est également mis en évidence dans la Figure B.7(b), où l'accélération

avec 32 processeurs est d'environ 10 lors de l'utilisation du BMM et autour de 8 lors de l'utilisation de l'API DMA et FIFO.



(a) (b)

FIGURE B.7: Comparaison des temps d'exécution (a) et speedup (b) entre le BMM et le DMA pour l'application SUSAN.

B.7 Conclusion

L'objectif de cette thèse était d'augmenter la programmabilité à travers la mise en œuvre d'une API logiciel standard pour la communication inter-processus, tout en réduisant les surcoûts de performance imposé par cette mise en œuvre et, par conséquent, améliorant les performances de communication.

La programmabilité des architectures multi-cœurs est améliorée par la mise en œuvre de MCAPI, qui spécifie des primitives de communication inter-processus. Ensuite, les surcoûts imposés par la charge logicielle ont été caractérisés et des mécanismes de communication pour la synchronisation et les transferts de données ont été proposés. Les résultats de performance obtenus montrent que les mécanismes proposés apportent des gains significatifs en termes de latence, débit, trafic réseau, temps de charge processeur et temps de communication.

Par conséquent, cette thèse a démontré que les mécanismes matériels développés en co-conception avec MCAPI peuvent réduire considérablement les surcoûts de performance imposés par une mise en œuvre logicielle. En outre, il est montré que les mécanismes peuvent être flexibles et facilement programmable.

Bibliography

[1] G.E. Moore. **Progress in digital integrated electronics**. In *International Electron Devices Meeting, 1975*, volume 21, pages 11–13, 1975.

[2] Herb Sutter. **The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software**, 2005. http://www.gotw.ca/publications/concurrency-ddj.htm [Accessed: 2016-01-05].

[3] A.M. Jallad and L.B. Mohammad. **Comparative analysis of middleware for multi-processor system-on-chip (MPSoC)**. In *9th International Conference on Innovations in Information Technology (IIT)*, pages 113–117, March 2013.

[4] C. Ebert and C. Jones. **Embedded Software: Facts, Figures, and Future**. *Computer*, 42(4):42–52, April 2009. ISSN 0018-9162. doi: 10.1109/MC.2009.118.

[5] J. Holt, A. Agarwal, S. Brehmer, M. Domeika, P. Griffin, and F. Schirrmeister. **Software Standards for the Multicore Era**. *Micro, IEEE*, 29(3):40–51, May 2009. ISSN 0272-1732.

[6] The Multicore Association. **Multicore Communications API**, 2011. http://www.multicoreassociation.org/workgroup/mcapi.php [Accessed: 2015-10-16].

[7] Mentor Graphics. **A Case for MCAPI: CPU-TO-CPU Communications in Multicore Designs**. White paper, Aug 2010.

[8] S. Miura, T. Hanawa, T. Boku, and M. Sato. **XMCAPI: Inter-core Communication Interface on Multi-chip Embedded Systems**. In *9th International Conference on Embedded and Ubiquitous Computing (EUC)*, pages 397–402, Oct 2011. doi: 10.1109/EUC.2011.78.

[9] L. Matilainen, E. Salminen, T.D. Hamalainen, and M. Hannikainen. **Multicore Communications API (MCAPI) implementation on an FPGA multiprocessor**. In *International Conference on Embedded Computer Systems (SAMOS)*, pages 286–293, July 2011. doi: 10.1109/SAMOS.2011.6045473.

[10] Lauri Matilainen, Erno Salminen, and Timo D. Hämäläinen. **MCAPI Abstraction on FPGA Based SoC Design**. In *Proceedings of the Annual FPGA Conference*, FPGAworld '12, pages 5:1–5:6, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1645-3. doi: 10.1145/2451636.2451641. URL http://doi.acm.org/10.1145/2451636.2451641.

[11] L. Matilainen, L. Lehtonen, J.-M. Maatta, E. Salminen, and T.D. Hamalainen. **System-on-Chip deployment with MCAPI abstraction and IP-XACT metadata**. In *International Conference on Embedded Computer Systems (SAMOS)*, pages 209–216, July 2012. doi: 10.1109/SAMOS.2012.6404176.

[12] C. Clauss, S. Pickartz, S. Lankes, and T. Bemmerl. **Towards a Multicore Communications API Implementation (MCAPI) for the Intel Single-Chip Cloud Computer (SCC)**. In *11th International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 148–155, June 2012. doi: 10.1109/ISPDC.2012.28.

[13] W.-T.J. Chan, A.B. Kahng, S. Nath, and I. Yamamoto. **The ITRS MPU and SOC system drivers: Calibration and implications for design-based equivalent scaling in the roadmap**. In *32nd IEEE International Conference on Computer Design (ICCD)*, pages 153–160, Oct 2014. doi: 10.1109/ICCD.2014.6974675.

[14] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, and T. Mattson. **A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS**. In *IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 108–109, Feb 2010. doi: 10.1109/ISSCC.2010.5434077.

[15] C. Jalier, D. Lattard, G. Sassatelli, P. Benoit, and L. Torres. **A Homogeneous MPSoC with Dynamic Task Mapping for Software Defined Radio**. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 345–350, July 2010. doi: 10.1109/ISVLSI.2010.110.

[16] ARM. **big.LITTLE Technology: The Future of Mobile**, 2013. http://www.arm.com/files/pdf/big_LITTLE_Technology_the_Futue_of_Mobile.pdf [Accessed: 2015-11-14].

[17] Nicholas Blachford. **Cell Architecture Explained Version 2**, 2005. http://www.blachford.info/computer/Cell/Cell1_v2.html [Accessed: 2015-11-14].

[18] Samsung Electronics Co., Ltd. **Heterogeneous Multi-Processing Solution of Exynos 5 Octa with ARM® big.LITTLE™ Technology**, 2012. http://www.arm.com/files/pdf/Heterogeneous_Multi_Processing_Solution_of_Exynos_5_Octa_with_ARM_bigLITTLE_Technology.pdf [Accessed: 2015-11-14].

[19] T. Limberg, M. Winter, M. Bimberg, R. Klemm, M. Tavares, H. Eisenreich, G. Ellguth, J. Schlussler, E. Matus, G. Fettweis, and H. Ahlendorf. **A Heterogeneous MPSOC with Hardware Supported Dynamic Task Scheduling for Software Defined Radio**. In *DAC/ISSCC Student Design Contest*, July 2009.

[20] F. Clermidy, C. Bernard, R. Lemaire, J. Martin, I. Miro-Panades, Y. Thonnart, P. Vivet, and N. Wehn. **MAGALI: A Network-on-Chip based multi-core system-on-chip for MIMO 4G SDR**. In *IEEE International Conference on IC Design and Technology (ICICDT)*, pages 74–77, June 2010. doi: 10.1109/ICICDT.2010.5510291.

[21] U. Ramacher, W. Raab, U. Hachmann, D. Langen, J. Berthold, R. Kramer, A. Schackow, C. Grassmann, M. Sauermann, P. Szreder, F. Capar, G. Obradovic, W. Xu, N. Bruls, Kang Lee, E. Weber, R. Kuhn, and J. Harrington. **Architecture and implementation of a Software-Defined Radio baseband processor**. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2193–2196, May 2011.

[22] T. Vander Aa, M. Palkovic, M. Hartmann, P. Raghavan, A. Dejonghe, and L. Van der Perre. **A multi-threaded coarse-grained array processor for wireless baseband**. In *IEEE 9th Symposium on Application Specific Processors (SASP)*, pages 102–107, June 2011. doi: 10.1109/SASP.2011.5941087.

[23] D. Lattard, E. Beigne, F. Clermidy, Y. Durand, R. Lemaire, P. Vivet, and F. Berens. **A Reconfigurable Baseband Platform Based on an Asynchronous Network-on-Chip**. *IEEE Journal of Solid-State Circuits*, 43(1):223–235, Jan 2008. ISSN 0018-9200. doi: 10.1109/JSSC.2007.909339.

[24] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. **ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix**. In Peter Y. K. Cheung and GeorgeA. Constantinides, editors, *Field Programmable Logic and Application*, volume 2778 of *Lecture Notes in Computer Science*, pages 61–70. Springer Berlin Heidelberg, 2003. ISBN 978-3-540-40822-2. doi: 10.1007/978-3-540-45234-8_7. URL http://dx.doi.org/10.1007/978-3-540-45234-8_7.

[25] R. Baert, E. Brockmeyer, S. Wuytack, and T.J. Ashby. **Exploring parallelizations of applications for MPSoC platforms using MPA**. In *Design, Automation Test in Europe Conference Exhibition*, pages 1148–1153, April 2009. doi: 10.1109/DATE.2009.5090836.

[26] F. Lemonnier, P. Millet, G.M. Almeida, M. Hubner, J. Becker, S. Pillement, O. Sentieys, M. Koedam, S. Sinha, K. Goossens, C. Piguet, M.-N. Morgan, and R. Lemaire. **Towards future adaptive multiprocessor systems-on-chip: An innovative approach for flexible architectures**. In *International Conference on Embedded Computer Systems (SAMOS)*, pages 228–235, July 2012.

[27] D. Melpignano, L. Benini, E. Flamand, B. Jego, T. Lepley, G. Haugou, F. Clermidy, and D. Dutoit. **Platform 2012, a many-core computing accelerator for embedded SoCs: Performance evaluation of visual analytics applications**. In *49th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1137–1142, June 2012.

[28] P. Mahr, C. Lorchner, H. Ishebabi, and C. Bobda. **SoC-MPI: A Flexible Message Passing Library for Multiprocessor Systems-on-Chips**. In *International Conference on Reconfigurable Computing and FPGAs*, pages 187–192, Dec 2008.

[29] **Message Passing Interface Forum**, 1994. http://www.mpi-forum.org/index.html [Accessed: 2015-11-16].

[30] D.L. Ly, M. Saldana, and P. Chow. **The challenges of using an embedded MPI for hardware-based processing nodes**. In *International Conference on Field-Programmable Technology*, pages 120–127, Dec 2009. doi: 10.1109/FPT.2009.5377688.

[31] Shih-Hao Hung, Wen-Long Yang, and Chia-Heng Tu. **Designing and Implementing a Portable, Efficient Inter-core Communication Scheme for Embedded Multicore Platforms**. In *IEEE 16th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 303–308, Aug 2010. doi: 10.1109/RTCSA.2010.17.

[32] A. Agbaria, Dong-In Kang, and K. Singh. **LMPI: MPI for heterogeneous embedded distributed systems**. In *12th International Conference on Parallel and Distributed Systems*, volume 1, pages 8 pp.–, 2006. doi: 10.1109/ICPADS.2006.56.

[33] A.M. Aji, J. Dinan, D. Buntinas, P. Balaji, Wu chun Feng, K.R. Bisset, and R. Thakur. **MPI-ACC: An Integrated and Extensible Approach to Data**

Movement in Accelerator-based Systems. In *IEEE 14th International Conference on High Performance Computing and Communication, IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS)*, pages 647–654, June 2012. doi: 10.1109/HPCC.2012.92.

[34] J.L. Abellan, J. Fernandez, and M.E. Acacio. **CellStats: A Tool to Evaluate the Basic Synchronization and Communication Operations of the Cell BE**. In *16th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pages 261–268, Feb 2008. doi: 10.1109/PDP.2008.49.

[35] Wajid Hassan Minhass, Johnny Öberg, and Ingo Sander. **Design and Implementation of a Plesiochronous Multi-core 4x4 Network-on-chip FPGA Platform with MPI HAL Support**. In *Proceedings of the 6th FPGAworld Conference*, FPGAworld '09, pages 52–57, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-879-7. doi: 10.1145/1667520.1667527. URL http://doi.acm.org/10.1145/1667520.1667527.

[36] André Nieuwland, Jeffrey Kang, OmPrakash Gangwal, Ramanathan Sethuraman, Natalino Busá, Kees Goossens, Rafael Peset Llopis, and Paul Lippens. **C-HEAP: A Heterogeneous Multi-Processor Architecture Template and Scalable and Flexible Protocol for the Design of Embedded Signal Processing Systems**. *Design Automation for Embedded Systems*, 7(3):233–270, 2002. ISSN 0929-5585. doi: 10.1023/A:1019782306621. URL http://dx.doi.org/10.1023/A%3A1019782306621.

[37] P. Francesco, P. Antonio, and P. Marchal. **Flexible hardware/software support for message passing on a distributed shared memory architecture**. In *Design, Automation and Test in Europe*, pages 736–741 Vol. 2, March 2005. doi: 10.1109/DATE.2005.156.

[38] J. Heisswolf, A. Zaib, A. Zwinkau, S. Kobbe, A. Weichslgartner, J. Teich, J. Henkel, G. Snelting, A. Herkersdorf, and J. Becker. **CAP: Communication aware programming**. In *51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2014. doi: 10.1145/2593069.2593103.

[39] OpenMP Architecture Review Board. **The OpenMP® API specification for parallel programming**, 1998. http://openmp.org/wp/ [Accessed: 2015-11-16].

[40] The Open MPI Project. **Open MPI: Open Source High Performance Computing**, 2004. http://www.open-mpi.org/ [Accessed: 2015-11-16].

[41] Khronos Group. **OpenCL™ (Open Computing Language)**, 2008. https://www.khronos.org/opencl/ [Accessed: 2015-11-16].

[42] Itseez. **OpenCV (Open Source Computer Vision)**, 2006. http://opencv.org/ [Accessed: 2015-11-16].

[43] Umakishore Ramachandran, M. Solomon, and M.K. Vernon. **Hardware support for interprocess communication**. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):318–329, Jul 1990. ISSN 1045-9219. doi: 10.1109/71.80159.

[44] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. **Active Messages: A Mechanism for Integrated Communication and Computation**. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, pages 256–266, New York, NY, USA, 1992. ACM. ISBN 0-89791-509-7. doi: 10.1145/139669.140382. URL http://doi.acm.org/10.1145/139669.140382.

[45] Kwan-Po Wong and Cho-Li Wang. **Push-Pull Messaging: a high-performance communication mechanism for commodity SMP clusters**. In *International Conference on Parallel Processing*, pages 12–19, 1999. doi: 10.1109/ICPP.1999.797383.

[46] R.B. Abdallah, T. Risset, A. Fraboulet, and J. Martin. **Virtual Machine for Software Defined Radio: Evaluating the Software VM Approach**. In *IEEE 10th International Conference on Computer and Information Technology*, pages 1970–1977, June 2010. doi: 10.1109/CIT.2010.334.

[47] J. Wassner, K. Zahn, and U. Dersch. **Hardware-software codesign of a tightly-coupled coprocessor for video content analysis**. In *IEEE Workshop on Signal Processing Systems*, pages 87–92, Oct 2010. doi: 10.1109/SIPS.2010.5624770.

[48] Shaoshan Liu and Jean-Luc Gaudiot. **Synchronization Mechanisms on Modern Multi-core Architectures**. In *Proceedings of the 12th Asia-Pacific Conference on Advances in Computer Systems Architecture*, ACSAC'07, pages 290–303, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 3-540-74308-1, 978-3-540-74308-8. URL http://dl.acm.org/citation.cfm?id=2392163.2392191.

[49] F. Calcado, S. Louise, V. David, and A. Merigot. **Efficient Use of Processing Cores on Heterogeneous Multicore Architecture**. In *International Conference on Complex, Intelligent and Software Intensive Systems*, pages 669–674, March 2009. doi: 10.1109/CISIS.2009.121.

[50] C. Kachris, G. Nikiforos, V. Papaefstathiou, Xiaojun Yang, S. Kavadias, and M. Katevenis. **Low-latency explicit communication and synchronization**

in scalable multi-core clusters. In *IEEE International Conference on Cluster Computing Workshops and Posters*, pages 1–4, Sept 2010. doi: 10.1109/CLUSTERWKSP.2010.5613092.

[51] J.C. Meyer and A.C. Elster. **Optimized Barriers for Heterogeneous Systems Using MPI**. In *IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 20–33, May 2011. doi: 10.1109/IPDPS.2011.124.

[52] Farhat Thabet, Yves Lhuillier, Caaliph Andriamisaina, Jean-Marc Philippe, and Raphael David. **An efficient and flexible hardware support for accelerating synchronization operations on the STHORM many-core architecture**. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 531–534, March 2013. doi: 10.7873/DATE.2013.119.

[53] P. Reble, C. Clauss, and S. Lankes. **One-sided communication and synchronization for non-coherent memory-coupled cores**. In *International Conference on High Performance Computing and Simulation (HPCS)*, pages 390–397, July 2013. doi: 10.1109/HPCSim.2013.6641445.

[54] Seung Hun Kim, Sang Hyong Lee, Minje Jun, Byunghoon Lee, Won Woo Ro, Eui-Young Chung, and J.-L. Gaudiot. **C!!-!!Lock : Energy Efficient Synchronization for Embedded Multicore Systems**. *IEEE Transactions on Computers*, 63 (8):1962–1974, Aug 2014. ISSN 0018-9340. doi: 10.1109/TC.2013.84.

[55] L. Papadopoulos, I. Walulya, P. Tsigas, D. Soudris, and B. Barry. **Evaluation of message passing synchronization algorithms in embedded systems**. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*, pages 282–289, July 2014. doi: 10.1109/SAMOS.2014.6893222.

[56] Sang-Il Han, A. Baghdadi, M. Bonaciu, Soo-Ik Chae, and A.A. Jerraya. **An efficient scalable and flexible data transfer architecture for multiprocessor SoC with massive distributed memory**. In *Proceedings of 41st Design Automation Conference*, pages 250–255, July 2004.

[57] D. Buono, T. De Matteis, G. Mencagli, and M. Vanneschi. **Optimizing message-passing on multicore architectures using hardware multi-threading**. In *22nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 262–270, Feb 2014. doi: 10.1109/PDP.2014.63.

[58] Shanyuan Gao, Bin Huang, and R. Sass. **The Impact of Hardware Communication on a Heterogeneous Computing System**. In *IEEE 21st Annual*

International Symposium on Field-Programmable Custom Computing Machines (FCCM), pages 234–234, April 2013. doi: 10.1109/FCCM.2013.43.

[59] S.S. Kumar, M.T.A. Djie, and R. van Leuken. **Low Overhead Message Passing for High Performance Many-Core Processors**. In *First International Symposium on Computing and Networking (CANDAR)*, pages 345–351, Dec 2013. doi: 10.1109/CANDAR.2013.62.

[60] S. Wallentowitz, M. Meyer, T. Wild, and A. Herkersdorf. **Accelerating collective communication in message passing on manycore System-on-Chip**. In *International Conference on Embedded Computer Systems (SAMOS)*, pages 9–16, July 2011. doi: 10.1109/SAMOS.2011.6045439.

[61] F. Clermidy, R. Lemaire, Y. Thonnart, and P. Vivet. **A Communication and configuration controller for NoC based reconfigurable data flow architecture**. In *3rd ACM/IEEE International Symposium on Networks-on-Chip, (NoCS)*, pages 153–162, May 2009. doi: 10.1109/NOCS.2009.5071463.

[62] C. Helmstetter, S. Basset, R. Lemaire, F. Clermidy, P. Vivet, M. Langevin, C. Pilkington, P. Paulin, and D. Fuin. **A dynamic stream link for efficient data flow control in NoC based heterogeneous MPSoC**. In *18th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 41–46, Jan 2013. doi: 10.1109/ASPDAC.2013.6509556.

[63] P. Burgio, A. Marongiu, R. Danilo, P. Coussy, and L. Benini. **Architecture and programming model support for efficient heterogeneous computing on tigthly-coupled shared-memory clusters**. In *Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pages 22–29, Oct 2013.

[64] P. Burgio, R. Danilo, A. Marongiu, P. Coussy, and L. Benini. **A tightly-coupled hardware controller to improve scalability and programmability of shared-memory heterogeneous clusters**. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 1–4, March 2014. doi: 10.7873/DATE.2014.038.

[65] Wei-Chun Ku, Shu-Hsuan Chou, Jui-Chin Chu, Chi-Lin Liu, Tien-Fu Chen, Jiun-In Guo, and Jinn-Shyan Wang. **VisoMT: A Collaborative Multithreading Multicore Processor for Multimedia Applications With a Fast Data Switching Mechanism**. *IEEE Transactions on Circuits and Systems for Video Technology*, 19(11):1633–1645, Nov 2009.

[66] Accellera Systems Initiative. **SystemC 2.2 and TLM 2.0**, 2007. http://accellera.org/downloads/standards/systemc/files [Accessed: 2015-11-10].

[67] OpenCores. **Plasma Version 3**, 2001. http://opencores.com/project,plasma,overview [Accessed: 2015-11-10].

[68] A.J. Pena and S.R. Alam. **Evaluation of Inter- and Intra-node Data Transfer Efficiencies between GPU Devices and their Impact on Scalable Applications**. In *13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 144–151, May 2013. doi: 10.1109/CCGrid.2013.15.

[69] Victor Frederico Silva, Cantidio de Oliveira Fontes, and Flavio Rech Wagner. **The impact of synchronization in message passing while scaling multicore MPSoC systems**. In *IEEE/IFIP 20th International Conference on VLSI and System-on-Chip, 2012 (VLSI-SoC)*, pages 267–270, Oct 2012. doi: 10.1109/VLSI-SoC.2012.7332114.

[70] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. **MiBench: A Free, Commercially Representative Embedded Benchmark Suite**. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7803-7315-4.

[71] Thomas Mesquida. **Portage de Benchmarks applicatifs sur architecture multi-coeur hétérogène**. Technical report, École Centrale de Lyon, 2015. Unpublished.

[72] E.W. Dijkstra. **A note on two problems in connexion with graphs**. *Numerische Mathematik*, 1(1):269–271, 1959. ISSN 0029-599X. doi: 10.1007/BF01386390. URL http://dx.doi.org/10.1007/BF01386390.

[73] Scott Pakin. **Receiver-initiated message passing over RDMA Networks**. In *IEEE International Symposium on Parallel and Distributed Processing, 2008*, pages 1–12, April 2008. doi: 10.1109/IPDPS.2008.4536262.